MASTER EN SCIENCES INFORMATIQUES

Les systèmes de détection d'intrusion: leçons à tirer d'un déploiement d'ASAX distribué dans un contexte universitaire

Vanderavero, Nicolas; Martin, Xavier

Award date:
2001

Facultés Universitaires Notre-Dame de la Paix, Namur

Institut d'Informatique

Année Académique 2000-2001

Les Systèmes de Détection d'Intrusion
-
Leçons à tirer d'un déploiement d'ASAX distribué dans un contexte universitaire

Nicolas Vanderavero & Xavier Martin

Mémoire présenté en vue de l'obtention du grade de Maître et Licencié en Informatique

Résumé

L'importance de la sécurité des systèmes informatiques a évolué de manière considérable ces dernière années. En effet, avec l'expansion du réseau Internet, les systèmes informatiques présentent leur faiblesses à un nombre croissant de pirates potentiels. Il est donc devenu nécessaire de se prémunir contre les agressions de ces derniers grâce à l'utilisation d'un système de détection d'intrusion.

Le système de détection d'intrusion ASAX est le résultat d'un projet de recherche dirigé par Baudouin Le Charlier. A la base développé pour effectuer des analyses sur un seul poste, sa flexibilité lui permit d'évoluer vers le statut d'application distribuée sur un ensemble d'hôtes. Quant au mécanisme de détection d'ASAX, il est basé sur des règles de type "condition $\rightarrow$ action" écrites dans un langage appelé RUSSEL.

Le but de notre stage à l'INSA de Rennes était de déployer la version distribuée d'ASAX dans un réseau du département informatique. Nous étudions dans ce mémoire la fonctionnalité d'ASAX dans le contexte d'une situation réelle. Malgré un degré de puissance élevé dans sa capacité de détection d'intrusions, la version distribuée d'ASAX souffre de quelques problèmes de flexibilité de l'implémentation et de facilité d'utilisation. Ce mémoire propose dès lors une évolution de l'architecture et des fonctionnalités fournies vers une version plus "clef en main" du logiciel, permettant ainsi à l'administrateur de configurer le système de manière transparente.

Abstract

The importance of security in information processing systems has evolved in a significant way these last years. Indeed, with the expansion of the Internet, a growing number of potential crackers could exploit the weaknesses of the information systems. It thus became necessary to guard the system against those aggressions. This has been possible by using an intrusion detection system (IDS).

The ASAX IDS is the result of a research project directed by Baudouin Le Charlier. Firstly developped to carry out analyses on a single station, its flexibility enabled it to evolve to the state of application distributed on a set of hosts. ASAX's mechanism of detection is based on rules of type "condition $\rightarrow$ action" written in a language called RUSSEL.

The goal of our training course at the INSA of Rennes was to deploy the distributed version of ASAX in a network of the computer science department. We studied in this master's thesis the functionalities of ASAX in a real context situation. In spite of a high degree of power in its capacity of intrusion detection, the distributed version of ASAX still suffers from some problems such as flexibility of the implementation and facility of use. This thesis proposes an evolution of the architecture towards a version more "on-the-road" of the software, making it thus possible for the administrator to configure the system in a fluent and transparent way.

Remerciements :

Nous désirons remercier le Professeur Baudouin Le Charlier pour le temps et la patience qu'il nous a consacrés afin de réaliser ce mémoire et la connaissance de qualité qu'il nous a transmise.

Un merci tout particulier à Mireille Ducassé, Véronique Abily et Jean-Philippe Pouzol qui nous ont accueillis à bras ouverts au sein de l'équipe de l'INSA. Un grand merci pour l'expérience que nous avons partagée ensemble et l'excellent contact que nous avons eu avec eux.

Un grand merci au Docteur Abdelaziz Mounji pour le temps qu'il nous a consacré à nous expliquer le fonctionnement d'ASAX distribué.

Nous souhaiterions remercier également Vincent Letocart, travaillant sur le projet ASAX à l'Institut d'Informatique de Namur, pour les conseils et avis techniques sur ASAX ainsi que pour l'aide pratique fournie sur l'utilisation de LaTeX.

Toute notre gratitude à Catherine Godest sans qui nous n'aurions pas trouvé de logement à Rennes pendant les 5 mois de notre stage.

Merci à nos parents et à tous nos amis qui nous ont soutenus moralement pendant la rédaction de ce mémoire.

Table des matières

Table des figures

Liste des Algorithmes

Introduction

Nous avons assisté ces dernières années à une expansion fulgurante du nombre de machines connectées en réseau et, plus particulièrement, à Internet. Il y a quelques années encore, ces connexions étaient réservées à quelques privilégiés issus du monde universitaire ou militaire. La technologie évoluant et les prix baissant, l'accès à Internet s'est banalisé et plus personne ne s'étonne de se le voir offrir à l'achat d'un ordinateur.

Cependant, la plupart de ces nouveaux utilisateurs d'Internet ne mesurent pas les conséquences que peut avoir la mise en réseau de leur ordinateur. Ils n'ont pas conscience que leur système d'exploitation et les logiciels qu'ils utilisent contiennent souvent de graves failles de sécurité. Ces failles, dues pour la plupart à des erreurs de programmation ou à une négligence de la part des concepteurs, peuvent être exploitées par des individus malveillants. Ils peuvent ainsi prendre le contrôle à distance d'un ordinateur ou encore le rendre inopérant.

Si l'augmentation des machines connectées a accru le nombre de victimes potentielles, les attaquants sont également devenus plus nombreux. L'actualité confirme l'ampleur de ce problème. Il ne se passe plus un mois sans que l'on ne parle d'un nouveau "ver" attaquant des serveurs web peu sécurisés ; d'une nouvelle attaque d'un pirate sur un site de vente en ligne ; d'un nouveau virus exploitant la crédulité des utilisateurs, ...

Les principales victimes sont les particuliers peu soucieux de la sécurité de leur machine, mais les entreprises ne sont pas pour autant à l'abri des "pirates". Les logiciels utilisés par ces entreprises sont en général plus sûrs, mais la fiabilité absolue n'existe pas.

En général, plus la fiabilité d'un système augmente, plus la liberté laissée aux utilisateurs diminue. On ne peut constater qu'avec regret que ces derniers perçoivent cette diminution de liberté comme un manque de convivialité. La convivialité d'utilisation est certes importante, mais il est inacceptable de la privilégier au détriment de la robustesse et de la sécurité des programmes.

Il devient de plus en plus difficile de transmettre un tant soit peu correctement des clefs d'analyse valables à un utilisateur exploité par la magie du marketing. Il se retrouve dès lors bien seul dans un univers cognitivement complexe où profit et vitesse sont les maîtres du jeu.

Afin de se protéger de certaines attaques, des outils ont été développés avec plus ou moins de succès. Citons par exemple les antivirus permettant de rechercher et d'éliminer les fichiers infectés ou encore les *firewalls* capables de contrôler et filtrer les connexions réseau effectuées sur une machine. Les systèmes de détection d'intrusion, ou IDS (Intrusion Detection System), se veulent être des outils généraux de protection capables de détecter les activités anormales au sein d'un réseau informatique.

Notre stage de fin d'études a porté sur l'utilisation d'un tel système de détection d'intrusion : ASAX. Le but de ce stage consistait à porter l'IDS sous Solaris 2.7 et à le déployer au sein du réseau du département informatique de l'INSA de Rennes. Ce mémoire a pour

objectif de présenter ASAX et le déploiement qui en a été effectué ; de mettre la lumière sur certaines limitations rencontrées et de proposer une extension de l'architecture du programme afin de surpasser ces dernières.

Sa structure est la suivante : le chapitre 1 présente les notions relatives à la détection d'intrusion. Nous y expliquons ce qu'est un IDS et présentons les différentes approches existantes. Nous passons ensuite en revue les différents types d'attaques ainsi que leur effets sur les machines qui en sont victimes. Nous décrivons ensuite les attaquants et tentons de connaître les motivations qui les poussent à effectuer de tels actes.

Le chapitre 2 présente l'architecture et le fonctionnement de l'IDS ASAX. Nous rappelons brièvement l'historique du projet puis décrivons la première version de ce produit. Nous nous attardons sur les 2 caractéristiques principales de cet IDS à savoir le langage RUSSEL et son format universel de description de données, le format NADF. Nous présentons ensuite la seconde version d'ASAX, nommée ASAX distribué. Cette version est capable, comme son nom l'indique, d'effectuer un processus de détection et d'analyse de manière distribuée.

Nous relatons, dans le chapitre 3, notre expérience du déploiement de la version distribuée d'ASAX au sein du département informatique de l'INSA de Rennes. Nous présentons tout d'abord l'organisation du déploiement au sein du réseau. Ensuite, nous expliquons les différents travaux effectués sur place : adaptation à l'environnement informatique, rédaction de règles de détection, intégration d'une source réseau dans l'analyse, ...

Au cours de l'utilisation et du portage d'ASAX, nous avons rencontrés quelques lacunes au niveau de son implémentation. Ces problèmes, ainsi que les solutions que nous avons adoptées, sont décrits au chapitre 4.

La version distribuée d'ASAX est toujours au stade de prototype et, bien qu'efficace, peut présenter un manque de souplesse dans son utilisation. Nous tentons donc, dans le chapitre 5, d'améliorer la flexibilité d'ASAX. Nous étudions d'abord les différents besoins de souplesse et de confort qu'un utilisateur est en droit d'attendre d'un IDS et proposons ensuite une extension de l'architecture d'ASAX distribué afin de satisfaire ces besoins.

Nous terminons, en abordant dans le sixième chapitre, la question de la sécurisation d'ASAX. Comme tout programme, celui-ci peut présenter des failles de sécurité. Nous lançons donc quelques idées de réflexions quant à l'amélioration de sa robustesse.

Chapitre 1

La détection d'intrusion

1.1 Introduction

Ce chapitre a pour but de familiariser le lecteur avec les techniques et les termes relatifs à la sécurité et plus particulièrement à la détection d'intrusion. On y définira la notion de système de détection d'intrusion et on y présentera les différentes approches existantes dans ce domaine. On explorera aussi l'autre côté du miroir en passant en revue les différents types d'attaques réalisables contre un système informatique et en précisant qui sont les attaquants et leurs buts.

1.2 Aperçu des systèmes de détection d'intrusion

En nous basant sur [SAN], nous définissons les notions relatives à la détection d'intrusion et nous classifions les systèmes de détection d'intrusion.

On peut considérer la détection d'intrusion comme "l'art de détecter les activités anormales, incorrectes ou inappropriées au sein d'un réseau informatique". Notons que les "intrusions" sont parfois distinguées des "abus[1]". Le terme "intrusion" désigne alors les attaques provenant uniquement de l'extérieur du réseau ; le terme "abus" concerne les attaques effectuées depuis l'intérieur du réseau. Cependant, dans la suite de ce document, nous utiliserons le terme "intrusion" pour désigner tant les attaques internes qu'externes.

Les systèmes de détection d'intrusion, ou IDS[2], sont donc des outils permettant de détecter ces activités malveillantes et éventuellement d'y mettre un terme avant qu'elles ne provoquent des dégâts.

1.2.1 Classification des systèmes de détection d'intrusion

Les IDS sont nombreux et variés et chacun d'eux, à sa manière, tente de résoudre le problème de la détection d'intrusion. On peut toutefois dégager deux "grands critères" qui permettent de classer les IDS :

- a) l'origine des données analysées
- b) le procédé utilisé pour effectuer l'analyse

[1] Misuse

[2] Intrusion Detection System

Si l'on considère l'origine des données analysées, on peut distinguer deux types d'IDS : les IDS *host-based* et les IDS *network-based* :

- Les IDS *host-based* analysent des événements se déroulant sur une machine. Dans la plupart des systèmes d'exploitation, tous les événements (exécution d'un programme, connexion d'un utilisateur, arrêt de la machine, ...) sont enregistrés dans des fichiers appelés *journaux des événements* (ou encore *données d'audit*, *audit trails* ou *logfiles*) grâce à un mécanisme d'audit. Les IDS *host-based* analysent donc ces journaux pour tenter de détecter toute activité malveillante.

  Ces *logfiles* étant inhérents à chaque machine, leur analyse sera en général effectuée par un processus propre à chaque ordinateur. En pratique, il faut donc que l'IDS soit installé et s'exécute correctement sur chaque machine du réseau à protéger.

- Les IDS *network-based* utilisent comme source de données le trafic circulant sur un segment d'un réseau pour détecter toute activité suspecte. Toutefois, afin de détecter un maximum d'attaques, il est judicieux d'effectuer l'analyse des "données réseaux" à plusieurs niveaux : analyse des en-têtes des paquets, de leur contenu, des corrélations entre différents paquets, ...

  Si les IDS n'effectuent pas une analyse en profondeur, on risque de passer à côté de beaucoup d'attaques.

Les approches *host-based* et *network-based* ne sont pas exclusives. L'étude menée dans [LHF+00] montre que les meilleurs IDS sont ceux combinant à la fois une analyse du "trafic réseau" et une analyse des données d'audit.

Si l'on considère le procédé utilisé pour effectuer l'analyse, on peut mettre en avant deux formes d'IDS : les IDS *knowledge-based* et les IDS *behavior-based* :

- Les IDS *knowledge-based* utilisent une "base de connaissances" contenant des informations sur différentes attaques et vulnérabilités. Ils analysent les données en les comparant avec cette base de connaissances. Si une tentative d'attaque est repérée, une alarme est déclenchée.

  En pratique, après comparaison avec la base de connaissance, toute action qui n'est pas considérée comme une attaque est autorisée. L'efficacité de ces IDS réside donc dans cette base de connaissances qui doit être mise à jour régulièrement afin de tenir compte des nouvelles attaques.

- Les IDS *behavior-based* tentent de découvrir des activités anormales en observant le comportement du système et des utilisateurs. Après une phase d'apprentissage, l'IDS dispose d'un modèle du "comportement normal" du système et de ses utilisateurs. Si l'IDS constate que l'activité courante s'éloigne trop du "comportement normal", il déclenche une alarme.

  Prenons un exemple pour illustrer ce principe : considérons un ensemble de dix machines destinées à un travail de bureau. Après observation, on considère que le "comportement normal" du système est d'être utilisé de huit heures du matin à cinq heures de l'après-midi. Le "comportement normal" des utilisateurs est de n'exécuter qu'un ensemble réduit d'applications, par exemple, un tableur et un traitement de texte. Si on exécute ultérieurement un compilateur C sur une des machines à trois heures du matin, l'IDS considère que l'utilisation du système s'écarte trop de la normale et déclenche alors une alarme.

  Les IDS *behavior-based* ont l'avantage de pouvoir détecter des tentatives d'exploitation de vulnérabilités jusque-là inconnues. Malheureusement, ces systèmes souffrent de deux désavantages importants :

- Si le système est attaqué alors que l'IDS se trouve en phase d'apprentissage, un "comportement agressif" sera considéré comme étant "normal".
- Le comportement d'un système et de ses utilisateurs n'est pas fixe. Il évolue au cours du temps et subit des variations saisonnières. Afin d'éviter de fausses alarmes, la phase d'apprentissage doit être assez longue afin d'intégrer les changements de comportements tout au long de l'année. Cependant, il n'est pas toujours possible d'autoriser une phase d'apprentissage aussi longue.

1.3 Classification et description des différents type d'attaques

Nous avons vu que le rôle principal d'un IDS est de détecter les attaques portées à une machine et, dans une certaine mesure, d'empêcher ces attaques d'aboutir.

Celles-ci peuvent avoir des buts différents et opérer à des niveaux divers ; certaines tentent d'exploiter des faiblesses ou des portes dérobées[3] dans le système d'exploitation ou les autres logiciels s'exécutant sur une machine. Certaines se basent sur les protocoles de communication utilisés sur le réseau dans lequel se trouve la machine. Enfin, d'autres requièrent un accès physique à la machine.

Afin de souligner cette diversité, nous présentons et commentons les grandes catégories d'attaques en reprenant la classification utilisée dans [Ken99] :

1.3.1 Le déni de service

Une attaque en "déni de service[4]" a pour but d'empêcher la machine attaquée de rendre les services qu'elle propose habituellement. L'attaquant essayera donc de "planter" purement et simplement le service ou d'en dégrader suffisamment les performances afin qu'il devienne inutilisable en pratique.

Un exemple très médiatisé d'une attaque DoS, entre autre relaté dans [Gug00], est la série d'attaques qui ont touché de nombreux serveurs de pages Web (Amazon.com, Buy.com, CNN, eBay, ...) en février 2000. Les sites les plus affectés par ce genre d'attaques sont bien évidemment les sites de vente en ligne. Hormis le coup que cela porte à l'image de marque des sociétés visées, on peut aisément imaginer les pertes qu'entraîne une inaccessibilité prolongée aux serveurs.

Il existe plusieurs façon de réaliser un DoS :

L'attaque logique

Lorsque l'attaquant effectue une attaque logique, il tente d'exploiter à distance une faille, c'est-à-dire une erreur de programmation dans un des programmes s'exécutant sur la machine cible. Les effets de ces attaques peuvent être divers : le programme vulnérable peut se terminer de manière fautive ou avoir un comportement aberrant ; la machine peut rebooter ou se bloquer, ...

Dans tous les cas, l'utilisation du programme ou même de la machine tout entière est rendue impossible tant que l'un ou l'autre n'est pas relancé.

[3] Backdoor
[4] DoS : Denial of Service

L'inondation

Dans le cas d'une attaque par inondation[5], l'attaquant tente d'épuiser très rapidement les ressources de la machine cible. En général, l'attaquant s'emploie à saturer la capacité de traitement de la victime en lui envoyant à un rythme élevé une multitude de requêtes. Saturée par une telle montée en charge, les performances se dégradent et la victime devient quasi sur-le-champ incapable d'assurer le bon fonctionnement des services qu'elle propose habituellement.

Récemment, afin d'augmenter l'efficacité des "inondations", certains attaquants ont effectué un déni de service de manière distribuée[6]. Dans ce cas, ce n'est plus une machine mais des centaines qui vont attaquer de manière coordonnée une même cible. Si la victime était capable de supporter la montée en charge générée par une seule attaque par *flooding*, elle a par contre très peu de chances de pouvoir résister au raz-de-marée provoqué par l'attaque simultanée de plusieurs machines.

Attaquer, une par une, des centaines de machines et ensuite les utiliser pour effectuer une attaque DDoS peut sembler irréaliste. Cependant, cette technique est devenue réalisable étant donné que de plus en plus de particuliers disposent d'une connexion permanente à Internet et ne se soucient guère de la sécurité de leur machine. Pour profiter de ces négligences, des outils ont été développés afin de rechercher des machines vulnérables et de les attaquer automatiquement. Une fois compromises, ces machines vont à leur tour rechercher d'autres hôtes vulnérables et en prendre le contrôle. Lorsque le nombre de machines compromises sera suffisant, elles pourront alors servir à effectuer une attaque distribuée.

Ainsi, on trouvera dans [Gib01] l'analyse d'une attaque DDoS où il apparaît qu'un seul attaquant utilisait près de 474 ordinateurs pour inonder sa cible.

Les virus et chevaux de Troie

Une autre menace pour les systèmes informatiques sont les tristement célèbres virus et chevaux de Troie.

Un virus informatique est un programme dont le but premier est de se reproduire en recherchant et parasitant d'autres programmes. Ce comportement est assez analogue au comportement des virus biologiques, d'où leur nom de virus informatique. Lorsqu'un virus infecte un programme, il copie son code dans le code du programme infecté. Toute exécution du programme infecté déclenche également une exécution de la copie du virus qui, à son tour, cherche et infecte d'autres programmes.

Hormis le fait qu'ils se reproduisent, ce qui en soit n'est pas extrêmement gênant, les virus sont également programmés pour effectuer des opérations destructives à un moment donné. Cette action destructive peut se déclencher à une date et heure données, par exemple tous les vendredi 13, ou suite à une action particulière de l'utilisateur. Leurs effets sont variés mais impliquent généralement une suppression d'une partie ou de tous les fichiers du système.

Un cheval de Troie est un programme dont le but final est de provoquer des dégâts et d'ouvrir des brèches de sécurité dans la machine sur lequel il s'exécute. Il consiste souvent en un programme d'apparence anodine : un jeu, un économiseur d'écran ou un petit utilitaire quelconque. Mais qu'on ne s'y trompe pas ; pendant que l'utilisateur pense s'amuser sur le dernier jeu à la mode, le programme révèle sa vraie nature et compromet le système. Contrairement à un virus, un cheval de Troie ne se reproduit pas et n'infecte dès lors pas

[5] Flooding
[6] DDoS : Distributed Denial Of Service

d'autres fichiers. Il est donc, et c'est là toute l'ironie du système, lancé volontairement par l'utilisateur qui a été trompé par son apparence anodine.

L'attaque physique

Il existe enfin une attaque très simple et très redoutable : l'attaque physique. L'attaquant possède ici un accès physique à la machine et il a tout le loisir de porter atteinte à son intégrité. Il s'agit en fait de vandalisme pur et simple, mais cela reste encore la meilleure façon de rendre une machine inutilisable. Ce genre d'attaque ne peut bien sûr être contrée par un logiciel et ceci souligne l'importance d'avoir une politique de sécurité globale intégrant un contrôle des accès physiques aux ordinateurs.

1.3.2 Obtention des privilèges d'administration à partir d'un compte utilisateur

Dans les attaques destinées à obtenir les privilèges d'administration à partir d'un compte utilisateur, un attaquant possédant déjà un compte d'accès sur une machine tente de se faire passer pour l'administrateur afin de pouvoir profiter de ses droits d'accès total à la machine.

Dans la plupart des systèmes d'exploitation multi-utilisateurs, les utilisateurs possèdent des permissions d'accès sur les fichiers. Par exemple, un utilisateur aura le contrôle total sur ses fichiers (droit d'accès en lecture, écriture et exécution) mais n'aura aucun droit sur les fichiers des autres utilisateurs, sauf si ceux-ci l'y ont expressément autorisé. De même, si la plupart des utilisateurs ont un accès en exécution sur les divers programmes du système, seuls quelques-uns sont autorisés à les modifier.

Parmi les utilisateurs d'une machine, il en est un particulier : c'est l'administrateur, aussi appelé *root* dans le monde Unix. Cette personne est, comme son nom l'indique, chargée d'administrer la machine et de veiller à son bon fonctionnement. Pour mener à bien sa mission, elle possède absolument tous les droits sur cette machine : elle peut ajouter ou supprimer des comptes d'accès ; modifier les mots de passe de ces comptes ; lire, supprimer ou modifier des fichiers d'autres utilisateurs ; ajouter de nouveaux programmes, modifier la configuration des programmes existants, ...

Obtenir un accès *root* sur une machine est donc particulièrement attrayant pour une personne mal intentionnée. De telles attaques sont souvent appelées des attaques "*user-to-root*".

Il y a plusieurs façons d'obtenir un accès *root* mais la plus connue est celle du *buffer overflow*. Dans cette technique, l'attaquant tentera d'exploiter une faille dans des programmes de type *SetUID Root*.

Lorsqu'un utilisateur lance un programme, celui-ci s'exécute en son nom et possède donc ses droits d'accès. Il est pourtant possible de déroger à cette règle en activant un attribut particulier d'un fichier exécutable : le bit *SetUID*. Lorsqu'un exécutable possède un tel attribut, celui-ci s'exécute avec les droits d'accès du propriétaire du programme et non pas avec les droits d'accès de la personne ayant exécuté ce programme. Bien évidemment, seul le propriétaire d'un programme peut décider d'activer ou de désactiver le bit *SetUID*. Les programmes appartenant à l'administrateur et possédant l'attribut *SetUID* sont appelés des programmes *SetUID Root*. Quand ils sont lancés par un utilisateur normal, ils s'exécutent avec les droits d'accès de l'administrateur. Clairement, les programmes *SetUID root* doivent être écrits avec la plus grande prudence ; comme ils s'exécutent avec des droits d'accès étendus, la moindre erreur de programmation peut avoir des conséquences fâcheuses.

L'attribut *SetUID* est très utile quand on désire partager certains de ses fichiers avec d'autres utilisateurs tout en contrôlant les actions effectuées sur ces fichiers. L'exemple le plus connu d'un programme *SetUID Root* est le programme *passwd* qui permet à un utilisateur de modifier son mot de passe. Lorsqu'un utilisateur se connecte et introduit son mot de passe, le système vérifie la validité de celui-ci grâce à des informations contenues dans le fichier `/etc/passwd`. Quand un utilisateur décide de modifier son mot de passe, il doit aussi modifier les informations contenues dans le fichier `/etc/passwd`. Cependant, il faut s'assurer de la validité des actions de l'utilisateur. En particulier, il faut vérifier qu'il ne modifie pas les informations concernant les mots de passe d'autres personnes. Il n'est donc pas acceptable d'autoriser l'accès en écriture à ce fichier pour tous les utilisateurs. De même, il n'est pas réaliste de laisser à l'administrateur l'exclusivité de l'accès en écriture à ce fichier. Chaque utilisateur désirant changer son mot de passe devrait alors contacter l'administrateur pour qu'il effectue personnellement le changement. C'est une perte de temps tant pour les utilisateurs que pour l'administrateur. On souhaite que les utilisateurs puissent modifier eux-mêmes ce fichier tout en contrôlant leurs actions. La solution adoptée est la suivante :

1. Les droits d'accès sur le fichier `/etc/passwd` sont modifiés pour que seul l'administrateur ait le droit de le modifier.
2. Le programme *passwd* est mis en place afin d'éviter de déranger l'administrateur chaque fois qu'une personne désire modifier son mot de passe. Ce programme *SetUID root* demande à l'utilisateur d'introduire son nouveau mot de passe et va ensuite modifier en conséquence le fichier `/etc/passwd`. Si le programme a été écrit correctement et s'il ne comporte pas de failles, on est alors certain que le programme ne modifiera que les informations de l'utilisateur concerné. Le but a donc été atteint : chacun peut modifier lui-même son mot de passe quand il le désire mais il est impossible à un individu mal intentionné de modifier le mot de passe des autres utilisateurs.

Les programmes *SetUID root* constituent une cible de choix pour un attaquant. Son but sera de modifier le cours normal de l'exécution et forcer ainsi ces programmes à exécuter une séquence d'instructions choisie par l'attaquant. En général, cette séquence d'instructions lancera une invite de commande[7] s'exécutant elle aussi avec les privilèges *root*. L'attaquant dispose alors d'un *shell root* et a donc le contrôle total sur la machine.

Les attaques en *buffer overflow* exploitent des erreurs de programmation ou du moins un manque de robustesse des programmes. Fréquemment, les programmes ont besoin de données qui doivent être entrées par l'utilisateur. Celui-ci peut les introduire au moyen d'un argument passé lors de l'exécution du programme. Cette entrée peut également s'effectuer via un fichier ou une invite de commande. Quand le programme lit ces données, il les copie dans une zone mémoire, un *buffer*, où il pourra les traiter à sa guise. Souvent, le programme s'attend à ce que les données respectent un certain format et aient une taille maximale. La taille du *buffer* accueillant ces données sera donc dépendante de la taille maximale des données.

Malheureusement, beaucoup de programmes font aveuglément confiance à l'utilisateur et ne vérifient pas le format et la taille des entrées avant de les copier dans le buffer. Il en résulte que, lorsque qu'un utilisateur malveillant introduit une donnée plus grande que la taille maximale prévue par le programme, la copie de cette donnée dans le *buffer* "déborde" (d'où le terme *buffer overflow*) et va alors écraser une partie du programme en mémoire. Quand l'attaque réussit, la partie du programme écrasée sera exécutée par le système. L'attaquant a alors atteint son but : le système exécute, avec les privilèges *root*, du code choisi par l'attaquant.

[7] Shell

Mettre au point une attaque en *buffer overflow* est loin d'être trivial ; il faut très bien connaître le fonctionnement du programme attaqué ; maîtriser l'architecture de la machine sur laquelle il s'exécute ; être capable d'écrire directement du code machine ; calculer les bons alignements en mémoire,... Nous invitons le lecteur intéressé par une explication détaillée du fonctionnement des attaques en *buffer overflow* à consulter [One96].

1.3.3 Obtention d'un accès sur une machine distante

Il est souvent nécessaire pour l'attaquant de posséder un compte utilisateur sur une machine. On a vu par exemple que les attaques en *buffer overflow* afin d'obtenir un accès *root* nécessitaient de disposer d'un tel compte. Les attaques permettant à une personne ne possédant pas de compte sur une machine d'en obtenir un, sont appelées des attaques "*remote-to-user*". Plusieurs techniques sont applicables :

Le buffer overflow

Les services proposés par des machines sont la plupart du temps gérés par des programmes s'exécutant avec des privilèges *root*. Si ces programmes présentent certaines vulnérabilités, l'attaquant peut à nouveau utiliser la technique du *buffer overflow* pour obtenir un accès sur cette machine. Pour ce faire, il va envoyer une requête bien choisie à un des services. Si le programme gérant le service est vulnérable, la requête aura pour effet d'altérer le déroulement normal du programme pour le forcer à exécuter un code choisi par l'attaquant, ledit code se trouvant à l'intérieur de la requête.

En général, le code choisi par l'attaquant aura pour effet de créer un nouvel utilisateur dans le système ou d'altérer le mot de passe d'un utilisateur existant afin que l'attaquant puisse utiliser le compte de cet utilisateur.

Le vol de mot de passe

Pour obtenir un compte d'accès sur une machine, le plus simple reste encore de récupérer le mot de passe d'un utilisateur et de se servir du compte qui y est associé. Il y a plusieurs façon de procéder :

- **Utilisation d'un dictionnaire de mots de passe**
  Les utilisateurs ont malheureusement tendance à choisir des mots de passe trop facile à deviner. Les noms communs, les dates d'anniversaire, les prénoms des enfants ou conjoint(e)s sont autant d'exemples de mauvais mots de passe. Il est très facile de se constituer un dictionnaire des mots de passe les plus courants. Un attaquant peut donc essayer ces mots de passe un à un et, avec un peu de chance et de temps, il découvrira éventuellement le bon mot de passe. Cela ne demande pas beaucoup d'efforts puisque que l'on trouve facilement des programmes qui automatisent cette tâche. Des dictionnaires de mots de passe usuels sont eux aussi aisément récupérables. Dans certains cas, ce n'est pas l'utilisateur qui est en faute, mais bien l'administrateur du système. En effet, à leur installation, la plupart des systèmes d'exploitations créent un ensemble de comptes utilisateurs standards auquel sont associés des mots de passe par défaut bien connus de tous. Il arrive malheureusement que des administrateurs ne prennent pas la peine de modifier ces mots de passe, laissant alors une porte grande ouverte pour les attaquants.

- **Espionnage des communications réseau**
  Lorsqu'on dispose d'un accès physique à un réseau informatique, il est possible d'écouter les données qui y transitent. En espionnant ainsi le réseau, il est possible de récupérer des mots de passe. En effet, lorsqu'un utilisateur s'identifie dans un système informatique en introduisant son nom et son mot de passe, ces données sont souvent envoyées sur le réseau "en clair", c'est-à-dire de manière non chiffrée. Les protocoles les plus répandus sont également les moins sécurisés. Ainsi, dans les protocoles `telnet` (connexion à distance), `POP3` (rapatriement du courrier électronique) et `FTP` (transfert de fichiers), le nom d'utilisateur et le mot de passe associé transitent en clair à travers le réseau et peuvent être dès lors interceptés par tout quidam.
- **Le *social engineering***
  Le *social engineering* regroupe toutes les techniques de récupération de mots de passe se basant sur les contacts sociaux et sur la naïveté de certains utilisateurs. Il y a beaucoup de pratiques différentes. Citons-en quelques unes à titre d'exemple :
  - L'attaquant peut téléphoner à la victime en se faisant passer pour l'administrateur système et lui signaler qu'il y a un problème avec le compte de cet utilisateur. Il lui demande ensuite de changer son mot de passe par un autre que l'attaquant aura lui-même choisi.
  - Inversement, l'attaquant peut se faire passer pour un utilisateur et téléphoner à l'administrateur système en prétextant qu'il a oublié son mot de passe et qu'il aimerait le récupérer.
  - Certaines personnes ont la très mauvaise habitude de noter leur mot de passe sur un bout de papier collé sur leur écran ou sur leur tiroir. Un rapide tour dans ces bureaux permet parfois de récupérer quelques mots de passe.

  En pratique, le *social engineering* est rarement une attaque technique. Il s'agit surtout d'avoir de bons talents de comédien pour récupérer des mots de passe sur un coup de bluff.

1.3.4 Les coups de sonde

Les coups de sonde ne sont pas des attaques proprement dites, mais sont souvent une étape nécessaire à leur préparation. Nous avons vu que pour réaliser la plupart des attaques, il faut connaître des informations sur le système cible. Par exemple, il est utile de connaître le numéro de version des programmes installés afin de vérifier s'ils comportent des faiblesses susceptibles d'être exploitées. Un attaquant essayera donc d'obtenir un maximum de renseignements tels que : la marque et la version du système d'exploitation ; la présence de certains services écoutant sur des numéros de ports bien connus ; le numéro de version des logiciels assurant ces services ; le nom des utilisateurs de la machine, ...

En ce qui concerne les services présents sur une machine, on peut effectuer du *port scanning*. Sur un serveur, les programmes assurant des services écoutent sur des ports, chaque port étant désigné par un numéro. L'utilisation des numéros de port est standardisée pour les services les plus répandus. Par exemple, un programme assurant un service de transfert via le protocole `FTP` écoute par défaut les requêtes sur le port 21.

Le *port scanning* consiste donc à tester les ports un par un en y envoyant une requête. Si une réponse est émise, alors on peut en déduire la présence d'un service. En analysant cette réponse et en envoyant éventuellement d'autres requêtes sur le port ainsi découvert, on peut obtenir des informations sur le programme assurant le service ou le système d'exploitation de la machine.

1.4 Qui sont les attaquants ?

On peut s'interroger sur l'identité des attaquants ainsi que sur les motifs qui poussent ces personnes à s'introduire frauduleusement dans un système informatique. Quels sont leur buts ? Que cherchent-ils ?

Avant toute chose, il convient de faire une distinction entre les **hackers** et les **crackers**, ces deux termes étant généralement confondus par les personnes[8] étrangères au monde de l'informatique.

Un hacker est une personne passionnée par ce qu'elle fait, qui maîtrise son matériel et veut l'exploiter au maximum de ses possibilités même si cela détourne le matériel de ses fonctionnalités originales. Le "matériel" dont il est question n'est pas forcément du matériel informatique. Le terme hacker peut s'appliquer aux informaticiens, aux électroniciens, aux musiciens,... A nos yeux, un hacker est un artiste qui maîtrise son art et tente d'exploiter son don au maximum. Il n'est nullement ici question de personne mal intentionnée ou de "cyber-criminel". Citons un extrait de la définition du terme hacker tirée de [Jar] :

> *An expert or enthusiast of any kind. One might be an astronomy hacker, for example. One who enjoys the intellectual challenge of creatively overcoming or circumventing limitations.*

Un cracker, quand à lui, est une personne qui passe son temps à pénétrer de manière frauduleuse dans des systèmes informatiques et qui, éventuellement, y provoque des dégâts. Ils sont fiers de leurs actes et n'hésitent pas à s'en vanter. On peut trouver dans [Vac] une description de ces individus :

> *On USENET, calling someone a "cracker" is an unambiguous statement that some person persistently gets his/her kicks from breaking from into other peoples computer systems, for a variety of reasons. S/He may pose some weak justification for doing this, usually along the lines of "because it's possible", but most probably does it for the "buzz" of doing something which is illicit/illegal, and to gain status amongst a peer group.*
> *Particularly antisocial crackers have a vandalistic streak, and delete filestores, crash machines, and trash running processes in pursuit of their "kicks".*

La différence entre un hacker et un cracker est merveilleusement bien expliquée dans [Ray01] :

> *The basic difference is this : hackers build things, crackers break them.*

Les motivations des crackers sont très variées : certains prennent le piratage informatique pour un jeu et testent ainsi leurs "aptitudes". Ces personnes ne font en général pas de dégâts sur le système attaqué. Ils tentent seulement de voir jusqu'où ils peuvent aller et essayent de cette manière d'améliorer leurs compétences.

D'autres, plus dangereux, sont animés par un désir de vengeance ou par le profit : vengeance contre un ex-employeur ; espionnage industriel ou même militaire,... D'autres encore sont motivés par leur convictions politiques ou religieuses et pirateront des sites Web fortement fréquentés afin de faire passer leur "message".

Enfin, il existe une catégorie à part, les *script kiddies*, qui n'ont pas de motivation particulière. Ce sont en général des adolescents, n'ayant aucune connaissance poussée en sécurité informatique, qui récupèrent sur Internet des scripts d'attaque et d'autres outils de piratage et qui les essayent sans réellement comprendre leur fonctionnement ou même

[8] et souvent par la presse grand public

leurs effets. Malheureusement, les dégâts provoqués par ces apprentis sorciers peuvent être importants.

1.5 Conclusion

Le décor est désormais planté : nous avons présenté les IDS, leurs rôles et leurs types. Nous avons également exposé les différents types d'attaques susceptibles d'affecter un système informatique, leurs effets ainsi que les différentes raisons qui animent les crackers.

Avant de poursuivre, nous pouvons déjà mettre en avant deux observations tirées de ce rapide tour d'horizon des IDS et des attaques :

Premièrement, on remarque que la facilité d'exécution de certaines attaques et les dégâts importants qu'elle peuvent occasionner rendent indispensable la présence d'outils de protection et détection, tels que les IDS, au sein des sites informatiques. Cependant, la conception et/ou la maintenance des IDS est loin d'être triviale vu le nombre croissant et la diversité des attaques.

Deuxièmement, il ne faut pas perdre de vue qu'un IDS seul ne constitue pas une protection suffisante. Si dans la suite de ce document nous étudierons exclusivement les IDS, ne perdons pas de vue que la sécurité informatique doit être appréhendée comme un problème global : les IDS devront donc s'intégrer dans une politique de sécurité mettant en oeuvre tant des IDS que des Pare-Feux[9] ou des antivirus.

Il faut aussi souligner l'importance de la mise à jour des logiciels ; ces mises à jour corrigent bien souvent des failles de sécurité et permettent ainsi de diminuer les possibilités d'attaques. Soulignons aussi que des protections logicielles seules ne sont pas suffisantes. Il est impératif de contrôler l'accès physique aux machines (par exemple, en limitant l'accès aux ordinateurs aux personnes détentrices d'un badge magnétique ou d'une carte à puce).

Enfin, il faut sensibiliser les utilisateurs à la problématique de la sécurité. Ceci implique qu'ils prennent conscience qu'un mot de passe est une donnée secrète et personnelle, qui ne se prête pas et qui ne se note pas sur un bout de papier laissé à la portée des regards de tout un chacun.

[9] Firewall

Chapitre 2

Présentation d'un IDS : ASAX

2.1 Introduction

Ce chapitre présente un exemple d'IDS : le programme ASAX. La section 2.2 présente le projet et son historique. Les section 2.3 et 2.4 décrivent respectivement la version monoposte et la version distribuée d'ASAX. Nous présentons également dans la section 2.3 les deux fondations d'ASAX, à savoir le format de données NADF et le langage RUSSEL.

2.2 Historique du projet ASAX

ASAX (Advanced Security Audit-trail Analysis on uniX) a été développé conjointement par l'Institut d'Informatique de l'Université de Namur et Siemens-Nixdorf Software S.A.

A l'origine, cet IDS a été conçu pour analyser les "audit trails" de deux systèmes d'exploitations de Siemens-Nixdorf : BS2000 et SINIX. Il faut souligner [Mou97] que le développement d'ASAX s'est effectué en parallèle avec le développement des mécanismes d'audit de BS2000 et de SINIX. Il n'y avait donc que très peu d'informations disponibles sur la sémantique et le format des données produites par les deux systèmes d'audit.

Historiquement, cette incertitude quant aux types de données à analyser a sans aucun doute fortement influencé le développement d'ASAX. C'est pourquoi, l'universalité était un des objectifs [HLCMM94, HLCMM92] majeurs à atteindre par ASAX. S'il est clairement destiné à être utilisé comme un IDS, sa conception, guidée par l'objectif d'universalité, lui permet d'être utilisé dans des domaines d'application autres que la détection d'intrusion. Nous reviendrons plus en détail par après sur les trois objectifs du projet : **universalité**, **puissance** et **efficacité**.

Si l'on reprend la classification présentée en 1.2.1, ASAX peut être considéré comme un IDS *host-based* et *knowledge-based*. Cependant, nous verrons que la grande flexibilité d'ASAX permet de le rendre facilement *network-based* ou encore hybride.

ASAX étant *knowledge-based*, la séquence des "événements système" produite par le mécanisme d'audit est analysée de manière séquentielle afin de découvrir si elle contient une sous-séquence d'événements correspondants à une attaque connue. Ce mécanisme de détection est basé sur des règles de type "condition $\rightarrow$ action" écrites dans le langage RUSSEL [HLCMM94] (RUle-baSed Sequence Evaluation Language).

Après le premier prototype, qui était une version "monoposte", une seconde mouture d'ASAX a été développée. Cette dernière est une version distribué [Mou97, MLCZH94]

capable de détecter des attaques réparties sur plusieurs machines. Nous allons maintenant détailler ces deux versions.

2.3 Architecture de la version monoposte d'ASAX

La première version d'ASAX est une version monoposte. Elle s'exécute sur une machine hôte et analyse uniquement des données situées sur cet hôte. Il n'y a donc aucun traitement distribué. Concrètement, un processus ASAX s'exécutant sur une machine hôte ne saura jamais qu'une autre machine du réseau exécute, elle aussi, ASAX. Il n'y aura donc jamais d'échange de données entre ces deux processus ASAX.

La figure 2.1 reprend l'architecture de la version monoposte. Nous pouvons y identifier deux composants principaux :

- **L'évaluateur**
  L'évaluateur est véritablement le cœur d'ASAX : il est responsable de l'analyse des données d'audit. Tout d'abord, il lit et compile un fichier source contenant le code RUSSEL des règles à appliquer lors de l'analyse. Ensuite, il attend de recevoir ses données d'audit.
  L'évaluateur possède deux particularités. En premier lieu, les données analysées par l'évaluateur sont traitées de manière séquentielle. L'absence de "retour en arrière" lors de la lecture des données est une des clefs de la rapidité de traitement d'ASAX . En second lieu, les données produites par le mécanisme d'audit du système d'exploitation ne sont pas lues directement par l'évaluateur. En effet, ce dernier n'accepte que des données sous un format propre à ASAX, le format NADF (Normalized Audit Data Format).
- **L'adaptateur de format**
  Le rôle de l'adaptateur de format est de récupérer les données natives produites par le mécanisme d'audit et de les convertir au format NADF afin que l'évaluateur puisse les analyser.

L'utilisation du format de représentation interne NADF et la séparation du programme en deux modules (l'évaluateur et l'adaptateur de format) procure à ASAX une relative indépendance par rapport à la source d'audit. En effet, si le format des données générées par le mécanisme d'audit vient à changer, il suffit de modifier uniquement l'adaptateur de format en conséquence, les autres composants du système ASAX restant inchangés.

En résumé, le fonctionnement d'ASAX tel qu'il est schématisé par la figure 2.1 se déroule de la manière suivante :

1. Le mécanisme d'audit enregistre les événements du système dans un **fichier d'audit**. Ce mécanisme d'audit ne fait pas partie d'ASAX. Il est en général fourni avec le système d'exploitation et son fonctionnement peut varier d'un système à l'autre. De ce fait, le format des données enregistrées dans le fichier d'audit varie lui aussi en fonction du système d'exploitation utilisé.
2. Le fichier d'audit est ensuite traité par l'**adaptateur de format** qui convertit les données au format NADF et les enregistre dans un fichier.
3. L'**évaluateur** a été lancé et initialisé. Il a donc lu et compilé les règles RUSSEL à utiliser. Dès que des données au format NADF sont disponibles, il va les lire et les analyser en y appliquant les règles RUSSEL définies par l'utilisateur.

Lors de son développement, il était prévu qu'ASAX effectue une analyse de l'audit de manière *off-line*. Dans ce mode de fonctionnement, les audits sont stockés sur disque et

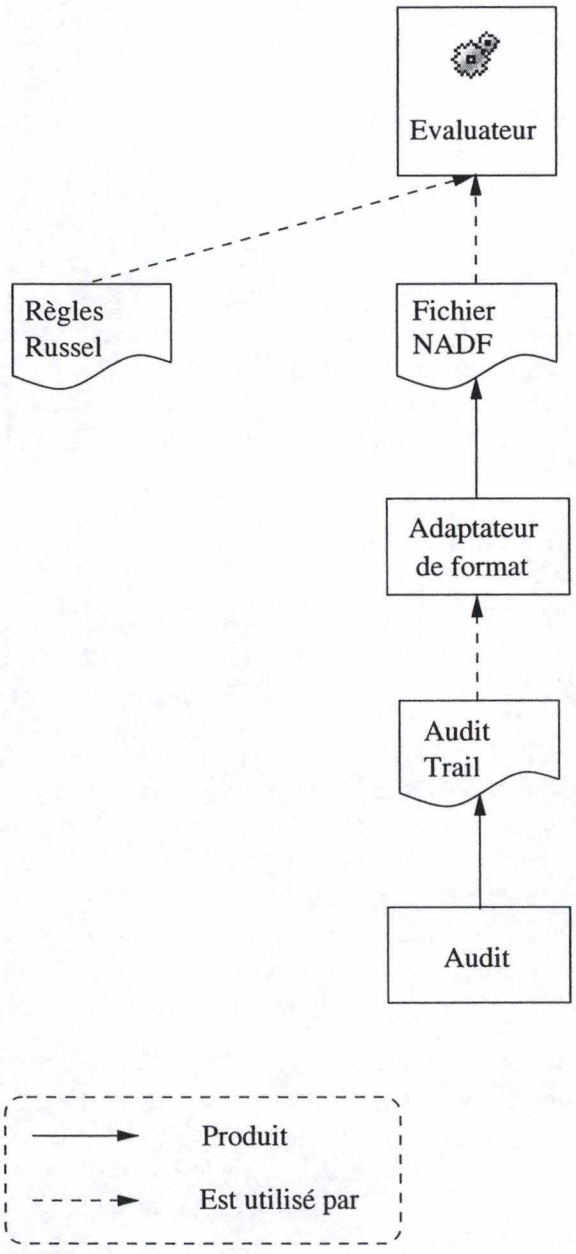


FIG. 2.1 – Architecture de la version monoposte d'ASAX

ne sont analysés que lorsque la charge du système le permet, par exemple durant la nuit, quand les utilisateurs n'effectuent plus de travaux. Pour autant que des fichiers NADF aient été préalablement stockés, il est donc possible d'exécuter l'évaluateur alors que le mécanisme d'audit ou même l'adaptateur de format n'ont pas été lancés. Cependant, l'efficacité d'ASAX est telle qu'il est capable d'analyser sans problème les fichiers NADF au fur et à mesure qu'ils sont générés par l'adaptateur de format. On utilisera donc ASAX pour effectuer une analyse continue en temps réel : l'audit enregistre continuellement des données qui, une fois converties à la volée par l'adaptateur de format, seront analysées directement par l'évaluateur.

2.3.1 Le format NADF

Nous venons de présenter le fonctionnement général d'ASAX. Nous allons à présent approfondir le premier pilier d'ASAX : le format NADF. Nous commençons d'abord par présenter la structure générale du format de données NADF et nous verrons ensuite pourquoi il est possible de traduire facilement des données d'audit quelconques vers ce format normalisé. La structure détaillée du format NADF sera ensuite présentée. Enfin, nous détaillerons la structure du fichier de description de données utilisé par l'évaluateur afin de pouvoir manipuler les données contenues dans les enregistrements NADF.

Principes généraux du format NADF

Le mécanisme d'audit enregistre les événements du système dans un fichier d'audit. Le format de ces données étant dépendant du système d'exploitation, nous les appellerons données *natives*, par opposition aux données NADF qui sont en quelque sorte *étrangères* au système d'exploitation.

Un fichier NADF est composé d'enregistrements et chaque enregistrement représente un événement qui s'est déroulé sur le système. L'ordre d'apparition des enregistrements dans le fichier NADF correspond à l'ordre chronologique des événements qui ont eu lieu sur le système. Chaque enregistrement est composé de champs représentant les différents attributs de l'événement. Par exemple, l'événement "ouverture de fichier" peut posséder des attributs tels que le nom du fichier ouvert, son chemin d'accès complet, le nom de son propriétaire, une indication de la réussite ou de l'échec de la tentative d'ouverture de ce fichier, un code d'erreur, ...

L'adaptateur de format a pour but de lire le fichier natif et de le traduire en fichier NADF. En général, les fichiers natifs sont agencés selon la même "philosophie" que les fichiers NADF : ils sont également composés d'enregistrements représentant chacun un événement. L'algorithmique de l'adaptateur de format est alors assez trivial : il suffit de lire séquentiellement les événements natifs et de les traduire un à un au format NADF.

Structure des enregistrements NADF

La figure 2.2 décrit la structure d'un enregistrement NADF : il est composé d'une entête *Longueur* de 4 octets et d'une suite de champs de tailles variables constituant le corps de l'enregistrement. L'en-tête *Longueur* indique la taille **totale** de l'enregistrement NADF (donc, cette en-tête y compris).

Chaque champ est composé de trois parties :

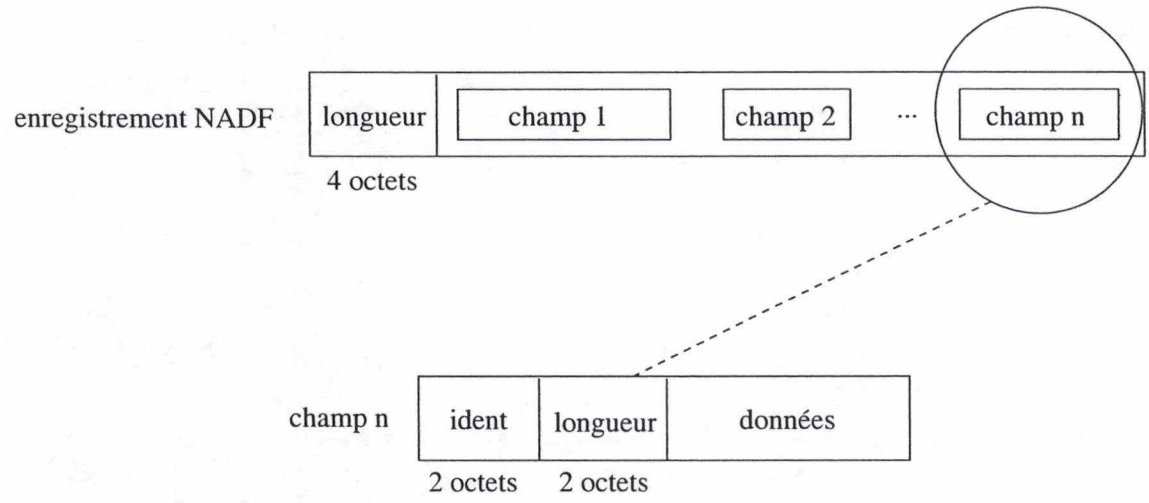


FIG. 2.2 – Structure d'un enregistrement NADF

1. Un identifiant NADF. C'est un entier non-signé codé sur 2 octets. Nous reparlerons du rôle de cet identifiant plus tard.
2. Un entier codé sur deux octets indiquant la taille des données du champ (donc, l'identifiant et l'entier indiquant la taille non compris).
3. Les données du champ en elles-mêmes.

Le format NADF impose certaines contraintes, notamment quant à l'alignement des données :

- Lorsque les enregistrements NADF sont stockés dans un fichier, ce dernier doit toujours commencer par une en-tête spécifique. Cette en-tête doit se conformer à la déclaration en C suivante :

```
struct {
        int len;
        char val[12];
} TypeNADF = {15,"__NADF__1| \0"};
```

- Les enregistrement NADF au sein d'un fichier NADF doivent toujours commencer à une adresse multiple de 4. Si nécessaire, des espaces (caractère dont le code ASCII est 32) seront rajoutées[1] à la fin d'un enregistrement afin que l'enregistrement suivant commence à une adresse multiple de 4. Les espaces ainsi rajoutées ne sont pas comptabilisées dans la taille totale de l'enregistrement.
- A l'intérieur d'un enregistrement NADF, les champs doivent commencer à une adresse paire par rapport au début de l'enregistrement. Si nécessaire, une espace sera rajoutée à la fin d'un champ pour respecter l'alignement. Cette espace n'est pas prise en compte dans le calcul de la taille des données du champ.
- Au sein d'un même enregistrement, les champs doivent apparaître selon l'ordre croissant des identifiants. Cette contrainte est imposée pour des raisons d'efficacité de l'évaluateur [HLCM94].

Notons enfin que le codage des informations au sein d'un enregistrement NADF est dépendant de la plate-forme où cet enregistrement a été généré. Il importe donc de tester

[1] En typographie, 'espace' est du genre féminin

l'en-tête du fichier NADF pour vérifier si l'on à affaire à un codage *Big-Endian* ou *Little-Endian.*

Le fichier de description de données

Revenons à présent sur le rôle des identifiants présents dans chaque champ NADF. A chaque identifiant correspond un et un seul nom, appelé nom de champ. Ce nom de champ permet de faire facilement référence à un champ quand on écrit le programme d'analyse. En effet, il est beaucoup plus facile pour le programmeur de manipuler des noms parlants plutôt que d'obscurs numéros identifiants.

Les relations *identifiant* $\leftrightarrow$ *nom de champ* sont définies dans un fichier réservé a cet effet et appelé fichier de description des données, ou encore Data Description File (DDF). Ce fichier sera utilisé par l'évaluateur afin de pouvoir faire la correspondance entre le nom de champ utilisé dans le programme RUSSEL et l'identifiant se trouvant dans l'enregistrement NADF.

Décrivons à présent sa structure.

Le DDF est un fichier texte décrivant les données pouvant se trouver au sein d'un fichier NADF. Chaque donnée est décrite par un bloc de 5 lignes contenant respectivement les informations suivantes :

- L'identifiant de la donnée : c'est le numéro identifiant qui est repris en en-tête d'un champ NADF. Il s'agit d'un entier compris entre 0 et 65535.
- Le type d'origine de la donnée : indique comment la donnée était interprétée dans le fichier d'audit (entier signé, non-signé, chaîne de caractères, ...)
- Le type de destination de la donnée : indique comment la donnée doit être interprétée par le programme d'analyse.
- Le nom identifiant de la donnée : c'est le nom de champ qui sera utilisé dans les programmes RUSSEL. Ce nom doit être différent d'un mot réservé de RUSSEL.
- Un texte libre décrivant la donnée.

Chacune de ces lignes commence respectivement par l'entier 1, 2, 3, 4 et 5. Notons que les lignes 2 et 3 décrivant respectivement le type d'origine et de destination de la donnée ne sont pas prises en compte lors de l'exécution d'un programme RUSSEL. En effet, en RUSSEL, toutes les données des champs NADF sont interprétées comme une "suite d'octets" sans autre type particulier.

2.3.2 Le langage RUSSEL

Après avoir vu l'architecture générale d'ASAX, nous avons examiné le format de données NADF ainsi que ses implications sur l'écriture des programme d'analyse, notamment au niveau des noms de champ. Nous passons maintenant à l'examen du deuxième pilier d'ASAX : le langage RUSSEL.

Nous commençons par expliquer les concepts de base de ce langage, à savoir : la notion de règles, leur exécution et leur application sur des enregistrements NADF. Après avoir exposé ces concepts de base, nous fixerons les idées en présentant un exemple de code RUSSEL et en simulant son exécution. Ceci permettra également au lecteur de se familiariser avec la syntaxe du langage. Nous présenterons finalement les fonctionnalités et instructions principales de ce langage.

Concepts de base

Un programme RUSSEL est composé de règles ; une règle pouvant être vue comme une procédure paramétrée classique. Nous verrons cependant que le mécanisme "d'appel de règle" est fort différent de "l'appel de procédure" présent dans les langages de programmation traditionnels. Une règle RUSSEL a pour but d'analyser les informations contenues dans un enregistrement NADF et, en fonction du résultat de l'analyse, d'effectuer différentes actions.

Les enregistrements NADF sont lus séquentiellement et vont être analysés un par un. Chaque fois qu'un enregistrement NADF est lu, il est analysé par un ensemble de règles appelé "ensemble des règles actives".

Au début de l'exécution du code, il n'y a qu'une seule règle active : la règle `init_action`. Cette règle particulière n'analyse pas d'enregistrement NADF ; elle est utilisée pour initialiser le processus d'analyse et pour définir quelles seront les règles actives lors de la lecture du premier enregistrement NADF. Une fois le premier enregistrement lu, les règles actives l'analysent. Notons qu'une règle ne peut modifier le contenu d'un enregistrement NADF. Les règles actives peuvent à leur tour activer d'autres règles qui analyseront cet enregistrement NADF courant ou définir des règles qui ne seront actives que lors de la lecture de l'enregistrement suivant. Par défaut, après avoir analysé un enregistrement, une règle qui était active ne l'est plus, sauf si elle est réactivée par elle-même ou par une autre règle.

On peut donc résumer le déroulement de l'analyse comme suit :

1. Exécuter la règle d'initialisation `init_action`. On utilisera cette règle pour initialiser des compteurs, activer les règles à appliquer à l'enregistrement NADF suivant (dans ce cas-ci, c'est le premier enregistrement) et définir les règles à appliquer à la fin de l'analyse, c'est-à-dire quand le dernier enregistrement a été lu et traité.
2. Tant qu'il y a des enregistrements à lire : lire un enregistrement et exécuter les règles actives.
3. Quand tous les enregistrements ont été lus et traités, exécuter les règles qui ont été désignées pour être actives à la fin de l'analyse.

Un exemple concret

Nous proposons maintenant de reprendre un exemple de programme RUSSEL tiré de [MLCM93]. Nous commenterons son exécution pas à pas et expliquerons la signification des différentes instructions.

Ce programme a pour but de comptabiliser et d'afficher le nombre de tentatives de connexions[2] manquées sur un ordinateur. Un *login* est considéré comme ayant échoué si l'utilisateur a introduit un mot de passe incorrect.

Le code source est repris à l'algorithme 2.1.

Le déroulement de l'exécution est le suivant :

La première ligne déclare la variable `count` comme étant une variable de type entier. La portée de cette variable est définie par la déclaration `global internal` indiquant qu'il s'agit d'une variable globale à ce fichier source. Nous verrons par après qu'il est possible de définir des variables globales dont la portée s'étend sur plusieurs "fichiers sources".

[2] Login

Algorithme 2.1 Code RUSSEL comptabilisant le nombre de *login* infructueux sur un hôte

```
1  global internal count : integer;
2
3  init_action;
4  begin
5          count := 0;
6          trigger off for_next count_bad_login;
7          trigger off at_completion echo_result
8  end;
9
10 rule count_bad_login;
11 begin
12         if
13           EVENT = 'login' and RESULT = 'failure' -> count := count+1
14         fi;
15         trigger off for_next count_bad_login
16 end;
17
18 rule echo_result;
19 begin
20         println(count,' bad login(s) found.')
21 end.
```

Avant de lire le premier enregistrement NADF, l'évaluateur exécute la règle `init_action`. La première instruction de cette règle est une affectation tout à fait classique : la variable `count` est initialisée à zéro. Les deux instructions `trigger off` suivantes sont des instructions d'activation de règles. La première, par l'utilisation du mot-clef `for_next` indique que la règle `count_bad_login` sera active lors de la lecture du prochain enregistrement. La seconde indique par le mot-clef `at_completion` que la règle `echo_result` sera active quand la lecture de tous les enregistrements sera terminée.

On voit donc par cet exemple qu'une règle n'est pas "appelée" comme l'est une procédure, mais est au contraire "activée". Lorsqu'une procédure en appelle une autre, l'exécution de la procédure appelante est suspendue et la procédure appelée est exécutée. Lorsque l'exécution de cette dernière est terminée, la procédure appelante reprend son exécution à l'instruction suivant directement l'instruction d'appel de procédure. Dans le cas d'un déclenchement de règle, l'exécution de la règle "appelante" n'est pas interrompue et la règle activée ne sera exécutée que lorsque l'enregistrement suivant sera lu (utilisation du mode `for_next`) ou lorsque tous les enregistrements auront été traités (mode `at_completion`). Les différents modes d'exécution seront décrits par après.

L'exécution de la règle `init_action` étant réalisée, l'évaluateur va lire le premier enregistrement NADF disponible et va exécuter l'ensemble de règles actives. Dans notre cas, seule la règle `count_bad_login` est active. La première instruction de cette règle est un test sur les valeurs des champs de l'enregistrement. Comme nous l'avons vu dans la section 2.3.1, les programmes RUSSEL manipulent des noms de champs plutôt que des identifiants NADF. La correspondance "identifiant $\leftrightarrow$ nom de champ" est définie dans le fichier DDF. Ici, les deux noms de champs sont `EVENT` et `RESULT`.

Le premier enregistrement va donc être analysé par l'instruction conditionnelle que l'on peut interpréter comme suit :

Si l'enregistrement NADF contient un champ dont le numéro identifiant correspond au numéro associé au nom de champ `EVENT`, que les données de ce champ sont la suite de caractères `login`, que l'enregistrement contient un champ dont le numéro identifiant correspond au numéro associé au nom de champ `RESULT` et pour autant que les données de ce champ sont la suite de caractères `failure`, alors la variable `count` est incrémentée d'une unité.

Par l'instruction suivante, la règle `count_bad_login` se réactive elle-même pour le prochain enregistrement. Si cette instruction avait été omise, la règle n'aurait analysé qu'un seul enregistrement et aurait été désactivée. L'évaluateur aurait ainsi continué à lire les enregistrements sans qu'aucune règle ne les analyse ; le programme se serait ensuite terminé.

Toutes les règles actives ayant analysé l'enregistrement courant, l'évaluateur lit l'enregistrement suivant et exécute les règles actives. Dans cet exemple, la règle `count_bad_login` est à nouveau la seule règle active.

Quand tous les enregistrements ont été lus et analysés, l'évaluateur exécute les règles qui ont été désignées pour être déclenchées à la fin de l'évaluation. Dans cet exemple, seule la règle `echo_result`, activée par l'instruction `trigger of for_next` de la ligne 7, sera exécutée. Elle a pour effet d'afficher la valeur de la variable `count`. Après cette ultime instruction, l'exécution du programme est terminée.

On le voit par cet exemple, l'écriture d'un programme RUSSEL ayant pour but de filtrer les événements d'un audit est assez simple.

Un exemple concret II

Donnons à présent un exemple de programme RUSSEL permettant d'effectuer une corrélation entre des événements grâce à l'utilisation de règles paramétrées.

Son but est de lever une alarme quand un utilisateur effectue la séquence d'actions suivante :

1. L'utilisateur utilise la commande `kill` pour tenter de tuer un processus qui ne lui appartient pas. Le système refuse d'exécuter la commande car il n'est pas le propriétaire de ce processus.
2. L'utilisateur exécute alors la commande `su`. Cette commande permet à un individu de devenir *root* pour autant qu'il connaisse le mot de passe de l'administrateur. Ne le connaissant pas, cette tentative échoue elle aussi.

Le code source est repris à l'algorithme 2.2.

Supposons un audit composé des événements suivants :

1. Le premier événement est une commande `kill` dont l'exécution a échoué. Cette commande a été lancée par l'utilisateur numéro 33333. Cet enregistrement contient donc les champs EVENT, RESULT et USER_ID ayant respectivement la valeur `kill`, `failure`, et 33333.
2. Le deuxième enregistrement est l'exécution fructueuse de la commande `kill` par l'utilisateur numéro 0. Par convention, l'utilisateur *root* porte le numéro 0.
3. Le troisième enregistrement concerne une exécution infructueuse de la commande `kill` par l'utilisateur numéro 1417.
4. L'enregistrement suivant indique que l'utilisateur numéro 1417 a lancé sans succès la commande `su`. Cet enregistrement contient donc les champs EVENT, RESULT et USER_ID ayant respectivement la valeur `su`, `failure`, et 1417.

Algorithme 2.2 Exemple de règles RUSSEL utilisant des paramètres

```
1  init_action;
2  begin
3  trigger off next watch_bad_kill
4  end;
5
6  rule watch_bad_kill;
7  begin
8          if
9            EVENT='kill' and RESULT='failure'
10                 -> trigger off next watch_bad_su(USER_ID)
11         fi;
12         trigger off for_next watch_bad_kill
13 end;
14
15 rule watch_bad_su(user : integer);
16 begin
17         if
18           EVENT='su' and RESULT='failure' and USER_ID=user
19                 -> begin
20                        println('Alerte sur l'utilisateur ',user);
21                        println('Tentative de terminaison d'un processus
                                suivie d'une tentative de passage en super-utilisateur')
22                 end;
23           true  -> begin
24                        trigger off for_next watch_bad_su(user)
25                 end
26 fi
27 end.
```

5. Le dernier enregistrement indique que l'utilisateur numéro 33333 a également lancé sans succès la commande `su`.

Simulons pas à pas l'exécution de programme sur cet audit.

1. La règle `init_action` est exécutée.
2. Le premier enregistrement est lu et est ensuite traité par la seule règle active : `watch_bad_kill`. Les informations contenues dans le premier enregistrement indiquent que l'utilisateur 33333 a effectué un `kill` infructueux. La règle active donc la règle `watch_bad_su` avec 33333 comme paramètre et se réactive elle-même.
3. L'enregistrement suivant est lu et est ensuite analysé par les deux règles actives : `watch_bad_kill` et `watch_bad_su(33333)`. L'événement contenu dans cet enregistrement étant une commande `kill` effectuée avec succès, la condition de l'instruction `if` de la règle `watch_bad_kill` n'est pas satisfaite. Elle se contente donc de se réactiver. Il en va de même pour la règle `watch_bad_su`.
4. Le troisième enregistrement est lu. Il n'y a toujours que deux règles actives : `watch_bad_kill` et `watch_bad_su(33333)`. Le test `if` de la règle `watch_bad_kill` étant satisfait, elle active une nouvelle instance de la règle `watch_bad_su` en lui passant comme paramètre le numéro d'utilisateur 1417. Ensuite, elle se réactive. La règle `watch_bad_su(33333)` quant à elle n'effectue pour toute action qu'une réactivation d'elle-même.
5. L'enregistrement suivant est lu. Les règles actives sont `watch_bad_kill`, `watch_bad_su(33333)` et `watch_bad_su(1417)`. L'événement contenu dans l'enregistrement est une exécution infructueuse, par l'utilisateur 1417, de la commande `su`. Les règles `watch_bad_kill` et `watch_bad_su(33333)` ne sont pas intéressées par cet événement et se réactivent. La règle `watch_bad_su(1417)` voit par contre la condition de son instruction `if` satisfaite. Elle déclenche donc une alarme et, son travail étant terminé, ne se réactive pas pour le prochain enregistrement.
6. Le dernier enregistrement est lu. Les deux règles actives sont `watch_bad_kill` et `watch_bad_su(33333)`. La règle `watch_bad_kill` analyse l'enregistrement courant. Sa condition `if` n'étant pas satisfaite, elle se contente de se réactiver. La règle `watch_bad_su(33333)` analyse elle aussi l'enregistrement courant. Cet enregistrement satisfaisant sa condition `if`, elle déclenche une alarme concernant l'utilisateur 33333. Elle ne se réactive pas.
7. Le dernier enregistrement ayant été lu et analysé, il faut encore exécuter les règles qui ont été activées par une instruction `trigger off at_completion`. Il n'y en a aucune. L'exécution du programme s'arrête donc ici.

Fonctionnalités et instructions principales

L'exemple de la comptabilisation des logins manqués a permis de présenter quelques éléments de la syntaxe de RUSSEL. Nous allons maintenant approfondir ce sujet et décrire les instructions les plus courantes. Le but n'est pas ici de rédiger un guide complet de la programmation en RUSSEL mais plus modestement de présenter les éléments caractéristiques de ce langage. Le lecteur pourra consulter [MLCM93, Mou97] pour obtenir une référence complète du langage RUSSEL ainsi que sa syntaxe au format BNF.

- **Les modules**
  Un programme d'analyse est composé d'un ensemble de règles RUSSEL. Pour faciliter l'écriture de ces programmes, on peut organiser ces règles en un ou plusieurs modules. On peut ainsi regrouper les règles portant sur un même domaine d'application en un

même module. Ce concept de "bibliothèque" est commun à bon nombre de langages et permet de faciliter l'écriture ainsi que la réutilisation du code.
Placée au début du code d'un module, l'instruction `uses` permet d'indiquer que ce module utilise des règles RUSSEL définies dans d'autres modules. Cette instruction est de la forme :

```
uses <nom_de_module_1>, <nom_de_module_2>, ..., <nom_de_module_n>
```

- **Les types de données**
  Le langage RUSSEL ne supporte que deux types de données : les entiers, notés `integer` et les chaînes d'octets, notées `string`. Le type entier désigne les entiers signés codés sur 4 octets. Il s'agit donc des entiers allant de -2^{31} à $2^{31} - 1$. Le type string désigne quant à lui une suite arbitraire d'octets.
  Les champ NADF sont considérés comme étant du type string. Considérer les champs comme étant une suite d'octet sans tenir compte de leur type d'origine peut sembler limitatif au premier abord. En réalité cette approche est correcte : les manipulations effectuées sur les champs des données d'audit se limitent la plupart du temps à des tests d'égalité ou des comparaisons du type 'plus petit que' ou 'plus grand que' qui se contentent de considérer les champs comme une "simple" suite d'octets sans autre type particulier. Cette démarche a en outre l'avantage d'éviter de coûteuses conversions de type. Cependant, il est parfois utile de pouvoir considérer les champs comme étant des valeurs de type entier : l'instruction `strToInt` permet donc d'évaluer un champ NADF, non pas comme un string, mais bien comme un entier.
- **Intégration de routines C**
  Il est possible d'intégrer dans l'évaluateur des routines programmées en C et de les appeler au sein d'un code RUSSEL. Une description complète de la marche à suivre pour intégrer des routines C peut être trouvée dans [MLCM93, Mou97]. ASAX contient déjà par défaut un ensemble de routines utilitaires. Nous en avons déjà rencontré deux exemples : `println` et `strToInt`. Ce n'étaient pas des instructions de base du langage RUSSEL mais bien des appel à des routines C.
  Cette possibilité d'ajouter des routines donne au langage RUSSEL une grande flexibilité et permet une personnalisation du langage en fonction du domaine d'application utilisé.
- **Les variables**
  Il existe deux types de variables : les variables locales et les variables globales.
  Les variables locales sont déclarées à l'intérieur d'une règle et ont une portée limitée à cette règle. La déclaration d'une variable locale à une règle se fait juste après la déclaration du nom de la règle et de ses paramètres et avant les instructions composant cette règle.

  Les variables globales doivent être déclarées au début d'un module. Une variable globale peut être externe ou interne. Si elle est déclarée comme interne, sa portée sera limitée au module dans lequel elle a été déclarée. Si elle est externe, elle sera connue de tous les modules constituant le programme d'analyse. Cependant, une variable globale externe doit être déclarée explicitement dans chaque module.
  La déclaration de variables globales internes est de la forme :

  ```
  global internal <var_1>, ..., <var_n> : <type>
  ```

  tandis que la déclaration de variables globales externes est de la forme :

  ```
  global external <var_1>, ..., <var_n> : <type>
  ```

- **Action conditionnelle**
  L'instruction `if` est une instruction d'action conditionnelle. Cette instruction est de la forme :

```
if
  condition_1 --> action_1
  ...
  condition_n --> action_n
fi
```

Elle exécute la première action, et uniquement celle-ci, pour laquelle la condition correspondante a été évaluée à la valeur "vraie".
Contrairement aux langages de programmation classiques, l'instruction `if` ne comporte pas de composante `else`. Celle-ci peut être facilement simulée en rajoutant une condition qui sera toujours évaluée à la valeur "vraie". Ainsi, le test `if condition_1 then action_1 else action_2` prendra la forme suivante en RUSSEL :

```
if
  condition_1 --> action_1
  true        --> action_2
fi
```

- **Action répétitive**
  L'instruction `while` est une instruction d'"action répétitive". Cette instruction est de la forme :

```
do
  condition_1 --> action_1
  ...
  condition_n --> action_n
od
```

  Son exécution est la suivante : la première action, et uniquement celle-ci, pour laquelle la condition correspondante a été évaluée à la valeur vraie est exécutée. Ensuite, l'exécution du bloc recommence jusqu'à ce que toutes les conditions soient devenues fausses.

- **Le déclenchement de règle**
  L'activation d'une règle s'effectue grâce à l'instruction `trigger off`. Sa syntaxe est de la forme :

  `trigger off <mode> <nom_de_la_règle>`

  où `<nom_de_la_règle>` est le nom de la règle à activer et `<mode>` est l'un des trois mode suivants : `for_next`, `for_current` ou `at_completion`.

  Le mode d'activation `for_next` indique que la règle sera active lors de l'analyse du prochain enregistrement NADF.
  Le mode `for_current` indique que la règle va être appliquée à l'enregistrement NADF courant.
  Le mode `at_completion` indique que la règle sera exécutée à la fin de l'analyse, lorsque tous les enregistrements NADF auront été lus et traités.

2.3.3 Les objectifs du projet

Après avoir décrit le fonctionnement et l'architecture générale du projet, nous pouvons à présent reprendre les objectifs du projet et, sur base de [Mou97, HLCMM94, HLCMM92], montrer comment ils ont été atteints :

- **Universalité**
  L'universalité est atteinte grâce à l'utilisation du format NADF. En effet, tous les audits trails existants peuvent être traduits dans ce format. L'analyse n'étant effectuée que sur des données au format NADF, ASAX peut donc s'adapter à toute source d'audit existante.

- **Puissance**
  La puissance est assurée par l'utilisation du langage RUSSEL qui permet d'exprimer des critères de sélection complexes sur des séquences arbitrairement longues d'événements. La section 2.3.2 de ce chapitre expose plus en détail le langage RUSSEL.

- **Efficacité**
  L'efficacité est un point critique dans l'élaboration d'un IDS. Qu'il soit *host-based* ou *network-based*, le flux de données à analyser par l'IDS peut rapidement devenir très important. Ce dernier sera donc soigneusement élaboré pour être à même d'analyser ce flux en temps réel.
  Dans le cas d'ASAX, l'efficacité est assurée, d'une part, par le principe sous-jacent de RUSSEL dans lequel chaque événement n'est jamais traité qu'une et une seule fois ; d'autre part, par une optimisation de l'évaluateur [HLCM94] soigneusement effectuée tant au point de vue des algorithmes que de l'implémentation.

2.4 Architecture de la version distribuée d'ASAX

Le prototype de la version distribuée [Mou97] d'ASAX a été réalisé par A.Mounji. Cette version a été développée spécifiquement pour s'exécuter sous le système d'exploitation Solaris 2.5 de SUN avec son mécanisme d'audit BSM (Basic Security Module) [Sun93]. Néanmoins, il est toujours possible d'adapter cette version d'ASAX à une autre source d'audit en écrivant un adaptateur de format approprié.

Dans cette version d'ASAX, l'analyse des données s'effectue de manière distribuée au sein du réseau : sur chaque machine, les évaluateurs analysent des données locales. On peut considérer cette analyse locale comme un filtre séparant les événements quelconques des événements susceptibles de rentrer dans un schéma d'attaque. Après ce "tri", les évaluateurs envoient ensuite les données pertinentes à un autre évaluateur, unique pour le réseau, appelé "évaluateur maître". Cet "évaluateur maître" analysera les données envoyées par ses "esclaves" afin d'y découvrir un schéma d'attaque.

Une analyse distribuée des audits permet d'établir des corrélations entre des événements se produisant sur des machines différentes. Il est alors possible de détecter des attaques qui seraient passées inaperçues avec la version monoposte d'ASAX. Illustrons notre propos par un exemple simplifié :

Sur un réseau de cinquante machines, un intrus tente de découvrir le mot de passe d'un utilisateur par essais et erreurs successifs. Une version d'ASAX monoposte s'exécute sur chacune des machines. La règle RUSSEL peut se résumer par : "si un utilisateur introduit trois fois de suite un mot de passe erroné, une alarme est déclenchée". L'intrus peut donc faire deux tentatives sur une machine pour ensuite passer à une autre et refaire deux tentatives. Il peut répéter le processus autant de fois qu'il y a de machines. Au bout du compte, il aura donc pu réaliser cent essais de mot de passe. On est bien loin de la limite des trois tentatives.

Avec la version distribuée d'ASAX, cette situation ne peut plus se produire. La règle RUSSEL appliquée sur chaque machine peut être interprétée comme suit : "si un utilisateur

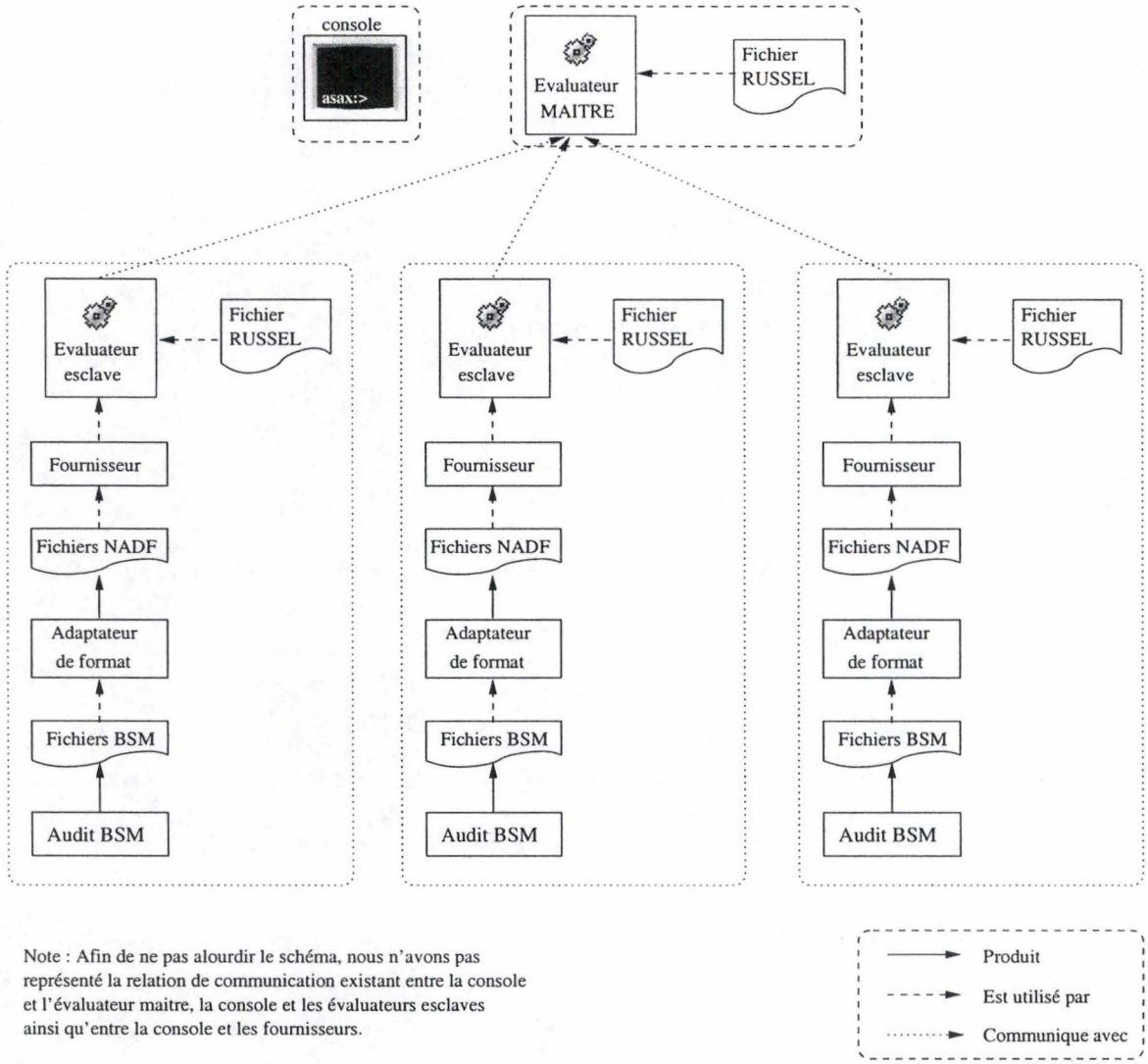


FIG. 2.3 – Architecture de la version distribuée d'ASAX

introduit un mauvais mot de passe, le signaler à l'évaluateur maître". La règle de l'évaluateur se résume par : "si je reçois, pour un même utilisateur, trois avertissements concernant un mauvais mot de passe, alors je déclenche une alarme". Pour que cet exemple soit applicable en pratique, il faudrait rajouter une condition sur le laps de temps écoulé entre chaque tentative. Mais cet exemple illustre bien l'intérêt d'effectuer une analyse distribuée.

Décrivons maintenant l'organisation et le fonctionnement des différents composants du système présentés à la figure 2.3. On peut remarquer trois types d'entités distinctes : une console, un maître et plusieurs esclaves.

- **La console**
  La console est le centre de commande du système ASAX. Elle permet à l'utilisateur d'avoir dans un simple terminal une vue complète du fonctionnement du système. Elle est responsable de l'exécution à distance des différents programmes composant le système ASAX. Elle permet également à l'utilisateur de reconfigurer à distance le comportement de certains exécutables.

- **L'évaluateur maître**
  Le maître est composé d'un évaluateur et de ses règles RUSSEL. Son fonctionnement est similaire à l'évaluateur de la version monoposte : il compile et exécute des règles

RUSSEL sur des données NADF. Cependant, si l'évaluateur de la version monoposte lisait les données depuis un fichier NADF, il en va tout autrement pour l'évaluateur maître ; ce sont les esclaves qui envoient des données NADF à l'évaluateur maître via le réseau.

- **Les systèmes esclaves**
  Le fonctionnement et la composition d'un esclave sont très proches d'un système ASAX monoposte. Nous pouvons cependant remarquer trois différences par rapport à l'ancien système. En premier lieu, le rôle de l'évaluateur est étendu. Il peut désormais envoyer des données au format NADF à l'évaluateur maître.
  En second lieu, le fichier NADF produit par l'adaptateur de format n'est pas lu directement par l'évaluateur esclave mais bien par un nouvel exécutable, nommé le fournisseur. Son rôle est de sélectionner, parmi plusieurs fichiers NADF correspondant à des audits produits à des moments différents, les événements appartenant à une période de temps déterminée. Cette période peut être configurée par l'utilisateur via la console.
  Enfin, signalons que l'adaptateur de format a également subi quelques modifications afin de faciliter l'archivage des fichiers d'audit natif ainsi que des fichiers NADF. Quand le fichier d'audit BSM lu par l'adaptateur de format dépasse une taille de trois méga-octets, ce dernier envoie une requête au mécanisme d'audit BSM. Cette dernière a pour effet d'arrêter l'enregistrement des données d'audit dans le fichier courant et de le continuer dans un nouveau fichier. De même, quand le fichier NADF produit par l'adaptateur de format atteint une taille de trois méga-octets, l'adaptateur le ferme et continue sa traduction dans un nouveau fichier. Ce "découpage en tranches de trois méga-octets" permet de déplacer et de sélectionner plus facilement les fichiers d'audit que l'on souhaite archiver.

Afin de fixer les idées, expliquons l'initialisation et le fonctionnement d'un système ASAX distribué tel que celui présenté à la figure 2.3.

Nous supposons que les différents exécutables ont été correctement installés et configurés sur chaque machine du réseau. De même, les fichiers contenant les règles RUSSEL à exécuter doivent être présents sur les différents hôtes. Enfin, le mécanisme d'audit et l'adaptateur de format doivent être lancés et opérationnels.

Une fois ces pré-requis satisfaits, l'utilisateur peut lancer la console et définir les paramètres de fonctionnement du système.

Cela est réalisé par le biais d'un fichier de description de l'environnement, appelé "fichier `edf`". Ce dernier permet de spécifier la machine maître d'une part et les machines esclaves d'autre part ; de spécifier les plages horaires que devront traiter les fournisseur. De plus, il indique, pour chaque évaluateur, le nom du fichier RUSSEL à exécuter.

La console lit ensuite ce fichier de description de l'environnement et lance à distance les différents composants du système : l'évaluateur maître, les évaluateurs esclaves et les fournisseurs associés. La console communique ensuite à ces éléments les paramètres nécessaires à leur initialisation ; les évaluateurs reçoivent le nom du fichier RUSSEL à analyser ; les évaluateurs esclaves prennent connaissance de l'adresse de l'évaluateur maître ; les fournisseurs sont informés de l'adresse de leur évaluateur.

Insistons sur le fait que la console reste en connexion avec tous ces éléments durant l'exécution de l'analyse distribuée. De cette manière, l'utilisateur peut à tout moment, via la console, envoyer des ordres aux composants du système. Ceux-ci peuvent en retour envoyer des messages d'information ou d'erreur qui seront affichés par la console.

Le système étant mis en route, les différents fournisseurs lisent leurs fichiers NADF

respectifs et en extraient les événements correspondant aux plages horaires déterminées par l'utilisateur.

Chaque fournisseur envoie alors ces données NADF à l'évaluateur esclave qui lui est associé. Ce dernier analyse ces données et, s'il le faut, envoie d'autres données au format NADF à l'évaluateur maître.

L'évaluateur maître collecte les données envoyées par les différents évaluateurs esclaves et les analyse.

Chapitre 3

Mise en oeuvre de la version distribuée d'ASAX : un cas réel

3.1 Introduction

Ce chapitre présente le déploiement et l'utilisation d'ASAX distribué tels qu'ils ont été réalisés durant notre stage de fin d'études. La section 3.2 décrit l'environnement général de notre stage ainsi que les objectifs à atteindre. Dans la section 3.3, nous précisons les différents travaux effectués afin d'adapter ASAX à l'environnement informatique qui était à notre disposition. Finalement, la section 3.4 présente les différentes applications pratiques que nous avons développées autour de l'IDS.

3.2 Présentation du projet

Dans le cadre de notre mémoire de fin d'études, nous avons été amenés à effectuer un stage au département informatique de l'INSA[1] de Rennes en France, sous l'encadrement du Professeur Mireille Ducassé, de Véronique Abily (l'administratrice réseau) et de Jean-Philippe Pouzol (doctorant).

La version distribuée d'ASAX présentée à la section 2.4 n'a jamais fait l'objet de tests au sein d'un réseau comportant un nombre élevé d'hôtes. Le but principal de ce stage était donc de déployer cette version d'ASAX sur le réseau du département informatique de l'INSA et de s'assurer de son bon fonctionnement. Pour cela, il fallait adapter ASAX à l'environnement du département, le configurer et l'installer. Ensuite, il était nécessaire de développer différentes règles de détection. Une fois ASAX rendu opérationnel, diverses mesures devaient être effectuées pour évaluer ses performances et sa consommation en ressources.

Le réseau du département est composé de deux sous-réseaux : le premier, le réseau *recherche*, connecte les différentes machines utilisées par le personnel scientifique et académique. Le second, le réseau *enseignement*, relie les différentes machines utilisées pour les cours ou les travaux pratiques. Le déploiement de la version distribuée d'ASAX devait se réaliser sur vingt-cinq machines du réseau *enseignement* : vingt-quatre machines en libre-service accessibles par les étudiants et une machine, nommée *bonnie*, utilisée comme serveur de fichiers et d'applications.

[1] Instituts Nationaux des Sciences Appliquées

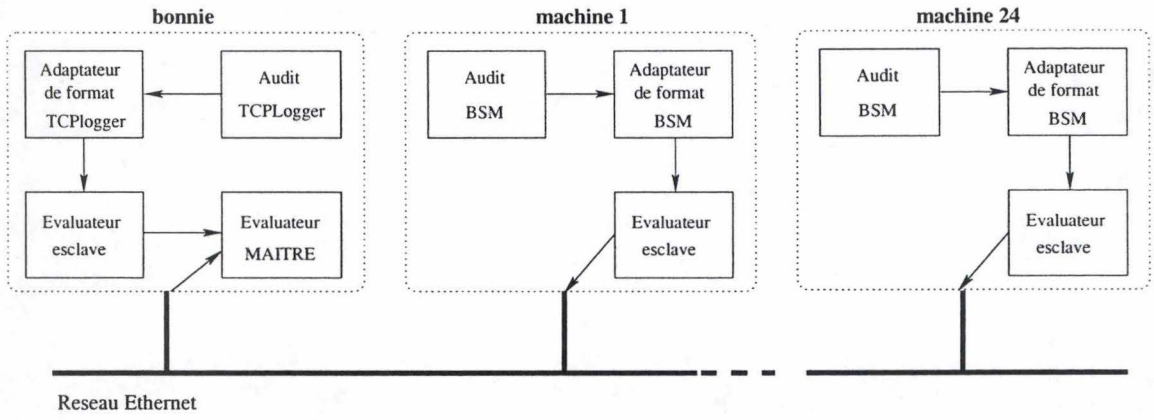


FIG. 3.1 – Architecture du déploiement d'ASAX distribué à l'INSA de Rennes

3.3 Déploiement au département informatique de l'INSA de Rennes

3.3.1 Définition de l'architecture de déploiement

En collaboration avec l'administratrice réseau, il a été décidé que les vingt-quatre machines en accès libre allaient chacune héberger un évaluateur ASAX esclave analysant les données fournies par l'audit BSM local. *Bonnie*, quant à elle, accueillerait la console et l'évaluateur maître. De plus, pour ne pas s'enfermer dans une approche uniquement *host-based*, il a été décidé de rajouter à l'IDS une source d'audit réseau. Notre choix s'est porté sur l'application *TCPlogger* faisant partie de la suite d'outils *Netlog* [NET94]. *TCPlogger* est une application capable de capturer les en-têtes des segments TCP circulant sur le réseau. C'est à nouveau *Bonnie* qui a été choisie pour accueillir l'évaluateur esclave analysant les données provenant de TCPlogger. L'architecture de déploiement que nous venons de décrire est résumée par la figure 3.1.

3.3.2 Portage et Déploiement

Nous dûmes commencer par une phase de portage et de modifications de l'application. En effet, le prototype de la version distribuée d'ASAX a été implémentée pour une station de travail SUN utilisant le système d'exploitation Solaris 2.5. Or, le réseau du département informatique de l'INSA est entièrement composé de stations utilisant Solaris 2.7.

Le code source d'ASAX ne pouvait malheureusement pas être compilé tel quel sous Solaris 2.7. Par rapport à Solaris 2.5, le format des données produites par le mécanisme d'audit BSM ainsi que quelques appels systèmes liés à cet audit sont en effet différents. De plus, cette version d'ASAX n'étant implémentée que pour utiliser BSM comme source d'audit, l'intégration de *TCPlogger* comme source d'audit supplémentaire impliquait également une révision du code.

Nous décrivons dans les sections suivantes les étapes accomplies pour porter ASAX distribué vers l'environnement du département informatique de l'INSA. Dans un premier temps, nous présentons les conséquences que peuvent provoquer un changement de source d'audit, ainsi que les impacts sur l'adaptateur de format. Ensuite, nous expliquons les modifications qui ont été apportées au fournisseur et à l'évaluateur afin d'améliorer l'ergonomie et la facilité d'utilisation. Ayant désormais plusieurs sources d'audit à notre disposition, il

est nécessaire d'avoir un mécanisme permettant de choisir laquelle utiliser. Nous proposons une modification qui permet d'effectuer ce choix.

3.3.3 Sources d'audit - Adaptateur de format

ASAX distribué était, dans le cadre de notre stage, destiné à être déployé au sein d'un sous-réseau de 25 machines. Il était donc nécessaire de commencer par installer et configurer BSM 2.7[2] sur toutes les machines de ce sous-réseau. Grâce à l'aide précieuse de l'administratrice système, cette opération a été effectuée sans problème.

Comme nous l'avons vu à la section 3.3.2, nous disposions comme point de départ d'une version d'ASAX distribué analysant une source d'audit BSM 2.5. Notre premier objectif était donc d'adapter cette version au format BSM 2.7. Pour y satisfaire, nous avons créé un nouvel adaptateur de format ; l'ancien n'étant plus valable. En effet, BSM, en passant d'une version à l'autre, ne propose plus la même structure de fichier d'audit. Par exemple, certaines incompatibilités sont apparues au niveau des noms des champs ainsi qu'au niveau de leurs tailles. Le code source de la fonction de conversion "BSM 2.7 $\rightarrow$ NADF" de ce nouvel adaptateur de format est repris dans l'annexe C.

Nous avons également développé un adaptateur de format pour l'audit réseau que nous avons choisi, *TCPlogger*. Le code source de cet adaptateur se trouve dans l'annexe D. Pour des raisons stratégiques, le rôle de cet adaptateur a été étendu à celui de fournisseur. En fait, il prend les données fournies par *TCPlogger* en entrée et communique directement la traduction au format NADF à l'évaluateur, sans passer par un stockage de cette dernière sous forme de fichier. Nous pouvons voir cette différence architecturale sur la figure 3.2.

ASAX dispose donc de deux sources d'audit différentes. L'idéal serait dès lors de les combiner de manière transparente en minimisant les ressources nécessaires à ce déploiement supplémentaire. Ce point est abordé à la section 3.3.5.

Nous avons également étudié la possibilité de créer un adaptateur de format pour l'outil SNORT [SNO]. Ce dernier est un analyseur de trafic réseau capable d'extraire les données contenues dans un paquet IP. Par rapport à TCPlogger qui ne récupérait que des en-têtes, SNORT peut extraire les contenus des messages. Cet outil permet dès lors d'effectuer des analyses plus complètes que celles envisageables avec TCPlogger. Mais SNORT n'est pas qu'un simple enregistreur de trafic réseau : il est également capable d'utiliser des règles afin de filtrer et de détecter des transmissions suspectes. Il a en outre l'avantage de déjà disposer d'un très grand nombre de règles qui sont mises à jour régulièrement et disponibles sur Internet. Cette abondance de règles ne peut que contenter l'administrateur système qui se voit ainsi dispensé de la fastidieuse tâche de conception de règles de détection. Concevoir un adaptateur de format pour SNORT ainsi qu'un traducteur de règles SNORT au format RUSSEL pourrait faciliter l'administration d'ASAX en tant qu'IDS *network-based*. Malheureusement, par faute de temps, nous n'avons pu développer d'adaptateur de format SNORT.

3.3.4 Fournisseur - Evaluateur

Nos travaux sur ces modules ne se rapportent pas au portage mais juste à quelques corrections de problèmes. Ces derniers sont relatés aux sections 4.2 et 4.3.

[2]Dans la suite de ce document, nous désignerons par BSM 2.7 et BSM 2.5 le mécanisme d'audit BSM utilisé respectivement par Solaris 2.7 et Solaris 2.5

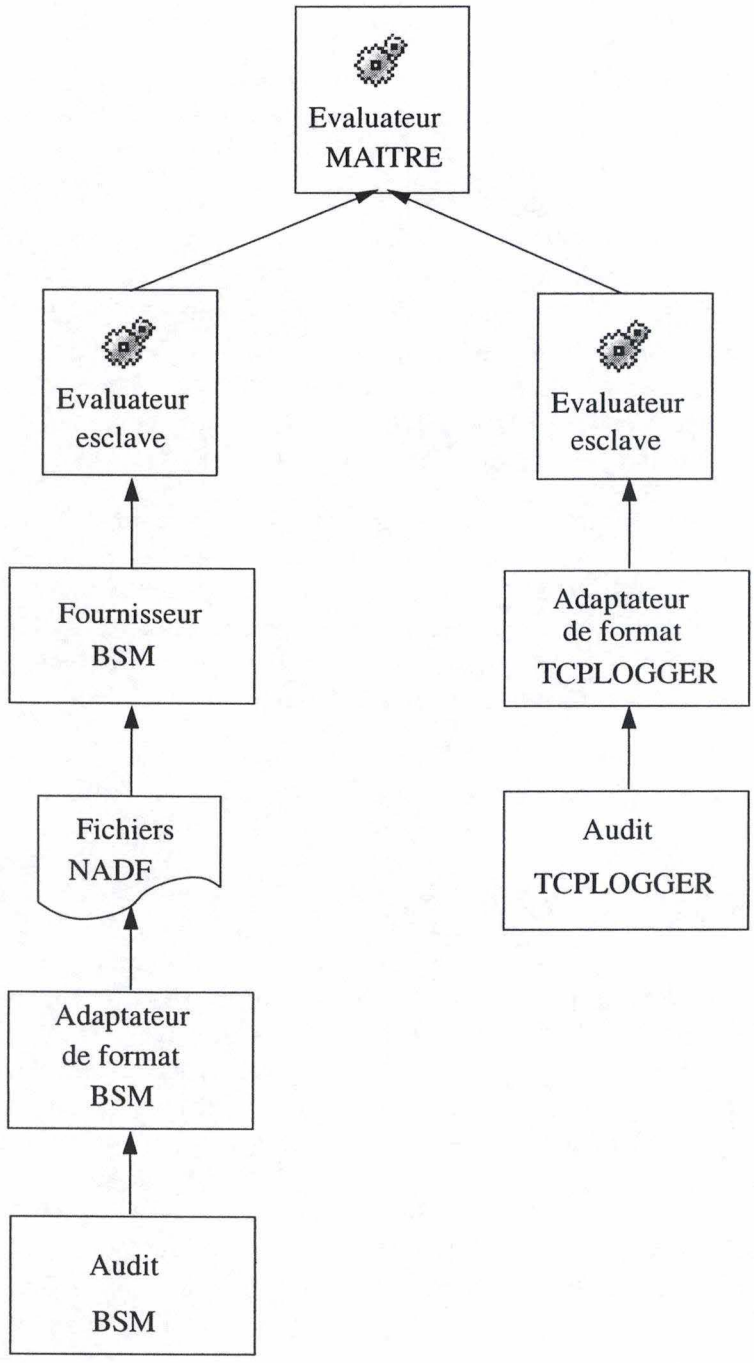


FIG. 3.2 – Différences architecturales entre l'évaluation de données provenant de l'audit BSM et TCPlogger

```
master machine_A : show;
slaves machine_B : send : bsm;
       machine_C : send : TCPlogger.
```

FIG. 3.3 – Choix du fournisseur correspondant à la source d'audit désirée dans le fichier descripteur d'environnement

3.3.5 La console

Quand la console lance un évaluateur à distance, elle lance également un fournisseur qui se connecte sur cet évaluateur. Nous avons déjà vu que l'adaptateur de format TCPlogger que nous avons développé ne stocke pas le résultat de sa conversion dans un fichier, mais se connecte directement à l'évaluateur pour lui fournir les données NADF. Il est donc en quelque sorte son propre fournisseur. Nous avons donc imaginé un moyen permettant de sélectionner, pour chaque évaluateur, le fournisseur que la console devra lancer à distance : le "fournisseur classique" ou l' "adaptateur-fournisseur TCPlogger". Pour cela, nous avons étendu la syntaxe des fichiers descriptifs d'un environnement ASAX distribué (fichiers `edf`).

La figure 3.3 illustre la nouvelle syntaxe de fichier descripteur d'environnement. On définit l'évaluateur maître, la machine sur laquelle il s'exécute et quelles règles RUSSEL il doit traiter. Il en est de même pour les évaluateurs esclaves, excepté le fait qu'ils sont chacun connectés à un fournisseur différent. Nous avons modifié le code de la console relatif au "parsing" du fichier descripteur d'environnement afin que l'on puisse choisir explicitement quel fournisseur utiliser. Il suffit d'indiquer le type du fournisseur (`BSM` ou `TCPlogger`) à la fin de chaque ligne correspondant à un évaluateur esclave.

Une fois que les évaluateurs ont été répartis sur les machines et que le type du fournisseur associé à chaque évaluateur a été choisi, l'architecture du réseau d'analyse restera en général fixe. Lorsque l'administrateur crée des fichiers `DDF`, il peut donc trouver fastidieux de devoir retaper le type du fournisseur associé à chaque évaluateur alors que cette association est quasi immuable. La solution que nous avons adoptée a été facilement mise en oeuvre, mais n'est pas la plus efficace du point de vue de la convivialité d'utilisation. Il serait peut-être bon, dans une version future, de prévoir un outil de génération de fichiers `DDF`. Un outil graphique permettant de créer et de configurer son réseau d'analyse serait une amélioration considérable à l'ergonomie d'ASAX.

Par exemple, dans la figure 3.3, l'évaluateur maître utilise les règles RUSSEL `show`. L'évaluateur esclave s'exécutant sur la machine_B utilise les règles `send` et est connecté à un fournisseur *BSM*. Celui de la machine_C utilise également les règles `send`, mais par contre, est connecté à un fournisseur *TCPlogger* (en fait directement à l'adaptateur de format *TCPlogger* vu la stratégie adoptée à la section 3.3.3).

Par la suite, nous avons apporté une seconde modification à la console. Celle-ci utilise le mécanisme `rexec()` pour lancer à distance les composants. Pour s'authentifier auprès de l'hôte distant, ce mécanisme a besoin du nom de l'utilisateur d'ASAX et de son mot de passe. Ces informations étaient codées en dur dans le code. Pour des raisons de sécurité évidentes, nous les avons enlevées du code et modifié la console. Désormais, la console demande le nom d'utilisateur et le mot de passe à son lancement. De cette manière, tout utilisateur connu dans le réseau peut utiliser ASAX distribué, si toutefois l'administrateur lui a accordé les permissions nécessaires.

3.3.6 Distribution du code d'ASAX

Le code du prototype d'ASAX distribué n'a pas été conçu pour être largement diffusé. Bon nombre de données décrivant l'environnement d'exécution sont codées en dur dans le code source. C'est le cas par exemple des chemins d'accès aux fichiers exécutables devant être lancés à distance. Pour l'utilisateur, cela signifie qu'il doit passer en revue le code source et modifier ces informations avant de lancer la compilation. Nous avons donc tenté de rendre l'installation d'ASAX plus conviviale.

Dans un premier temps, nous avons modifié les sources pour que les informations concernant l'environnement ne soient plus codées en dur. Elles sont désormais passées en paramètre lors de la compilation. Le gain pour l'utilisateur est appréciable ; il ne doit plus modifier les fichiers sources un par un mais doit juste adapter un seul fichier : le *Makefile*. Ce fichier décrit l'ordre de compilation des fichiers, les paramètres à passer au compilateur ainsi que d'autres informations permettant d'automatiser la compilation des différents fichiers sources.

Dans un deuxième temps, nous avons imaginé d'utiliser un script pour automatiser la modification du *Makefile*. Plutôt que de créer un tel script de toute pièce, nous avons préféré utiliser les outils GNU `automake` [MT98] et `autoconf` [ME98]. Ces outils permettent de générer d'une part un canevas d'un fichier *Makefile* et d'autre part un script de configuration. Quand ce script est exécuté, il analyse la configuration de la machine. En se basant sur cette analyse, il crée ensuite un *Makefile* à partir du canevas.

L'utilisation de ces outils permet de créer une distribution d'un code source possédant une procédure de configuration, de compilation et d'installation standardisée et quasi automatique. Cette procédure se limite en général à l'introduction de ces trois commandes :

```
$ configure
$ make
# make install
```

La commande `configure` invoque le script de configuration. La commande `make` compile les différents fichiers sources sur base du *Makefile* généré par `configure`. La commande `make install` copie les exécutables dans leurs répertoires respectifs et nécessite les droits de *root*.

Nous avons donc maintenant une archive qui peut s'installer plus facilement. Cependant, les outils `automake` et `autoconf` ne sont pas encore exploités au maximum de leur potentiel. Ainsi, l'archive créée ne contient que le code d'ASAX distribué adapté à Solaris 2.7. Une amélioration possible consiste à intégrer le code original pour Solaris 2.5 dans l'archive et de générer un script de configuration qui, en fonction de la version du système d'exploitation, compile le code adéquat. Ce script peut également détecter automatiquement l'architecture de la machine (Big-Endian ou Little-Endian) et compiler le code en fonction de cette architecture. De même, il ne faudra compiler et installer que les adaptateurs de format correspondant aux mécanismes d'audit effectivement installés sur la machine.

3.4 Applications

Après avoir décrit le travail réalisé pour porter et adapter le prototype de la version distribuée d'ASAX, nous décrivons les diverses applications effectuées au sein d'un environnement ASAX entièrement configuré et opérationnel.

3.4.1 Elaboration d'un traceur d'événements en langage RUSSEL

Afin de tester l'IDS, il importait de trouver des attaques, d'écrire des règles de détection pour ces attaques et d'analyser le comportement de l'IDS. Il existe de nombreux scénarios d'attaque disponibles sur Internet. Cette publication de scénarios est très utile car elle incite à se tenir au courant des nouvelles vulnérabilités découvertes et permet à l'administrateur système de tester la robustesse de son réseau.

Pour détecter les attaques, nous devons d'abord en avoir une caractérisation permettant de les identifier. Cette caractérisation consiste en une séquence d'événements appelée "scénario" ou "signature d'attaque" [KNMH00].

Pour connaître la séquence d'événements produits par un programme effectuant une attaque, il faut "tracer" les événements qu'il génère. Pour cela, il existe un outil nommé *truss* [TRU]. Il affiche à l'écran tous les appels systèmes effectués par le programme qu'il trace. Malheureusement, il n'est pas aisé de créer des règles RUSSEL à partir des informations fournies par `truss`. En effet, la personne écrivant des règles a l'habitude de manipuler les noms de champs définis dans le fichier `DDF`. En général, ces noms sont proches des noms des champs BSM correspondants. Or, les appels systèmes représentés par `truss`, respectent une dénomination et un format qui ne correspondent en rien aux conventions utilisées par l'audit BSM. Il est donc assez pénible d'écrire une règle de détection sur base des informations renvoyées par `truss`.

Nous avons donc décidé de créer un traceur en langage RUSSEL. Ce traceur utilise ASAX pour capturer tous les événements BSM générés par un programme donné. Il affiche ensuite ces événements selon la nomenclature utilisée dans le fichier `DDF` relatif à Solaris 2.7.

Remarquons que ce traceur a été écrit "à la main" et n'a pas été généré à partir du fichier `DDF`. Si ce dernier venait à être modifié, il convient de reporter ces modifications dans le code RUSSEL du traceur.

Remarquons également que le traceur se base sur la traduction NADF des événements enregistrés par le mécanisme d'audit BSM. Si ce dernier est mal configuré ou est configuré pour n'enregistrer qu'une fraction des événements système, la trace produite risque de ne pas être exhaustive.

Expliquons à présent le fonctionnement du traceur RUSSEL. Quand un programme est lancé, le système lui attribue un numéro identifiant : le PID[3]. Ce numéro reste fixe durant toute l'exécution du programme mais peut varier d'une exécution à l'autre du même programme. Lorsqu'un événement BSM est enregistré, il contient toujours un champ indiquant le PID du programme ayant généré cet événement. Il est donc aisé d'écrire un programme RUSSEL affichant tous les événements créés par un processus ayant un PID donné.

Cependant, un problème pratique se pose. Comme le PID change à chaque exécution, il faut donc lancer le programme à tracer, récupérer son PID, modifier les règles RUSSEL en fonction de ce PID, puis seulement lancer l'analyse. Cette façon de procéder est trop complexe et, dans le cas d'une analyse à la volée, ne permet pas de capturer les événements générés entre le lancement du programme et le lancement de l'analyse.

Ce problème a été résolu grâce à l'utilisation d'un programme intermédiaire. Nous avons écrit un programme, nommé `track`, prenant en paramètre le nom d'un autre programme ou d'une ligne de commande, et l'exécutant via la commande C `system`. Nous avons ensuite écrit une règle RUSSEL recherchant l'événement "exécution du programme dont le nom est `track`". Cette règle attend ensuite que le programme effectue l'appel à la fonction `system`.

[3]Process ID

```
                        ------------------
asaxd:Evaluator(2)@bonnie | Process: 29842 |
asaxd:Evaluator(2)@bonnie ------------------
asaxd:Evaluator(2)@bonnie header32_event_ID : 213 (munmap)
asaxd:Evaluator(2)@bonnie header32_date     : Wed Jan 17 15:58:14 2001
asaxd:Evaluator(2)@bonnie - - - - - - - - - - - - - - - - - - - - - - - - -
asaxd:Evaluator(2)@bonnie subject32_audit_ID      : 4230
asaxd:Evaluator(2)@bonnie subject32_user_ID       : 4230
asaxd:Evaluator(2)@bonnie subject32_group_ID      : 30185
asaxd:Evaluator(2)@bonnie subject32_real_user_ID  : 4230
asaxd:Evaluator(2)@bonnie subject32_real_group_ID : 30185
asaxd:Evaluator(2)@bonnie subject32_process_ID    : 29842
asaxd:Evaluator(2)@bonnie subject32_session_ID    : 968
asaxd:Evaluator(2)@bonnie subject32_device_ID     : 6291462
asaxd:Evaluator(2)@bonnie subject32_machine_ID    : 194.199.184.226
                                                    (balanec.insa-rennes.fr)
asaxd:Evaluator(2)@bonnie - - - - - - - - - - - - - - - - - - - - - - - - -
asaxd:Evaluator(2)@bonnie return32_process_error : 0
asaxd:Evaluator(2)@bonnie return32_process_value : 0
asaxd:Evaluator(2)@bonnie - - - - - - - - - - - - - - - - - - - - - - - - -
asaxd:Evaluator(2)@bonnie arg32_0_argument_value : -277348352
asaxd:Evaluator(2)@bonnie arg32_0_text           : addr
asaxd:Evaluator(2)@bonnie - - - - - - - - - - - - - - - - - - - - - - - - -
asaxd:Evaluator(2)@bonnie arg32_1_argument_value : 4096
asaxd:Evaluator(2)@bonnie arg32_1_text           : len
asaxd:Evaluator(2)@bonnie - - - - - - - - - - - - - - - - - - - - - - - - -
```

FIG. 3.4 – Exemple d'affichage d'un événement capturé par le traceur RUSSEL

Elle détecte cet événement et récupère le PID du programme lancé par `track`. Il ne reste plus qu'à passer ce PID en paramètre à la règle affichant tous les événements générés par un programme ayant un PID donné.

Pour tracer un programme, il faut donc lancer l'analyse avec la règle RUSSEL de traçage puis lancer le programme `track` avec comme paramètre le nom du programme à tracer. Si le programme tracé crée un processus fils, notre traceur le détecte et s'appelle récursivement afin de tracer l'exécution de ce nouveau processus.

Sans le développement de ce traceur, nous ne serions jamais arrivés à trouver les signatures des attaques décrites dans la section 3.4.2. Un exemple d'événement affiché par notre traceur est illustré à la figure 3.4.

Le code RUSSEL de notre traceur se situe à l'annexe A.1.

3.4.2 Développement de règles RUSSEL

Grâce à notre traceur, nous avons établi avec précision quelques signatures d'attaques. Nous avons ensuite codé des règles RUSSEL capables de détecter toute occurrence de ces signatures. La suite de cette section présente les différents cas traités. Pour chaque règle, nous expliquons brièvement l'attaque effectuée ainsi que les principes utilisés pour la détecter. Ensuite, pour tester l'efficacité de la détection, nous donnons un court jeu de test.

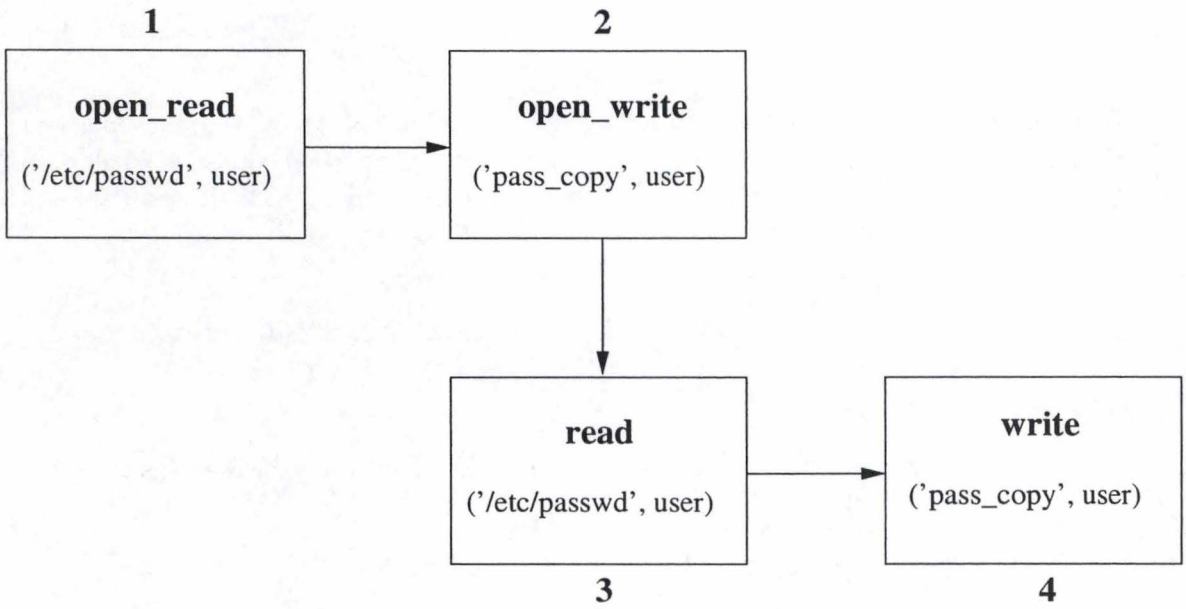


FIG. 3.5 – Détection d'une copie du fichier */etc/passwd*

Détection d'une copie du fichier */etc/passwd*

Explication de l'attaque

La première règle développée détecte la copie du fichier `/etc/passwd`. Cette détection est basée sur les événements générés pas le programme GNU de copie de fichier `cp`. Il est évident que toute copie de ce fichier doit lever une alerte. En effet, un utilisateur peut en effectuer une copie et la rapatrier ensuite sur un des ses ordinateurs personnels. Il peut alors tranquillement tenter de casser les informations contenues dans le fichier `/etc/passwd` en utilisant une attaque du type attaque par dictionnaire telle qu'elle a été présenté à la section 1.3.3.

Principe de la détection

La signature est composée de quatre événements à détecter (illustré à la figure 3.5) :

1. Ouverture en lecture du fichier source `/etc/passwd`
2. Ouverture en écriture du fichier destination `pass_copy`
3. Lecture du fichier `/etc/passwd`
4. Ecriture dans le fichier destination `pass_copy`

Tout événement est accompagné de l'identité UID[4] de l'utilisateur l'ayant provoqué. Cette donnée sert d'élément corrélateur entre les 4 événements du scénario. Donc, lors de la détection de l'ouverture du fichier `/etc/passwd`, on note le UID fourni de manière à pouvoir corréler les événements ultérieurs avec la personne suspecte. Si nous ne tenons pas compte de cet élément, le scénario suivant lèverait une fausse alarme :

1. Bernard lit le fichier `/etc/passwd`
2. Martine écrit dans un fichier personnel

[4]user ID

Ces deux personnes sont tout à fait dans leur droit ; il n'y a donc pas lieu de constater un comportement douteux.

Remarquons que nous n'essayons pas de détecter l'exécution du programme *cp* avec le fichier `/etc/passwd` comme paramètre d'exécution. En effet, rien n'empêche l'utilisateur malicieux de copier ce programme sous un autre nom, par exemple *tagada*. Il ne lui resterait plus qu'à l'exécuter comme suit :

```
$ tagada /etc/passwd pass_copy
```

et ASAX n'y aurait vu que du feu étant donné qu'il serait à l'écoute de l'exécution de *cp*.

Test de la détection

Les procédures de test de la validité des règles RUSSEL sont répertoriées dans le tableau suivant :

Commandes	Détection
`cp /etc/passwd pass_copy`	OUI
`cp /tmp/toto pass_copy`	NON

Le code source des règles RUSSEL se trouve à la section A.2.1.

Détection d'un vol de shell

Explication de l'attaque

La deuxième attaque traitée est le "vol de shell". Elle consiste à "voler" le shell d'un utilisateur afin de pouvoir effectuer des actions en son nom. La démarche de l'attaque est la suivante :

1. L'attaquant, appelons-le X, se fait passer par un moyen quelconque pour l'utilisateur A. Par exemple, il suffit que ce dernier s'absente 5 minutes sans avoir verrouillé sa station. X peut dès lors en profiter et prendre sa place devant l'écran.
2. Il copie un shell (par exemple `/bin/sh`) et affecte à la copie le *SetUID bit*. Rappelons que le rôle du *SetUID bit* a été expliqué à la section 1.3.2. Il s'assure ensuite que la copie est placée dans un répertoire sur lequel il possède des droits d'accès.
3. Cela fait, X n'a plus besoin d'attendre que A quitte son poste de travail pour pouvoir effectuer des actions en son nom : il lui suffit désormais d'exécuter la copie du shell.

Principe de la détection

Nous ne pouvons évidemment pas détecter la première étape qui consiste à utiliser la machine d'une personne en son absence. De plus, ne connaissant pas tous les moyens existants de détourner une machine, cette détection ne serait pas exhaustive. Nous nous concentrons donc sur le comportement suspect qu'est la copie d'un shell et l'affectation du *SetUID bit* à cette copie. L'affectation s'effectue via la commande `chmod` avec `4755` comme paramètre. Ce groupe de chiffres signifie que tout utilisateur pourra exécuter la copie, et que l'exécution se déroulera sous le nom de A.

La détection de la signature se fait en deux temps :

1. détection de la copie de `/bin/sh`
2. détection de l'application du *SetUID bit* sur la copie

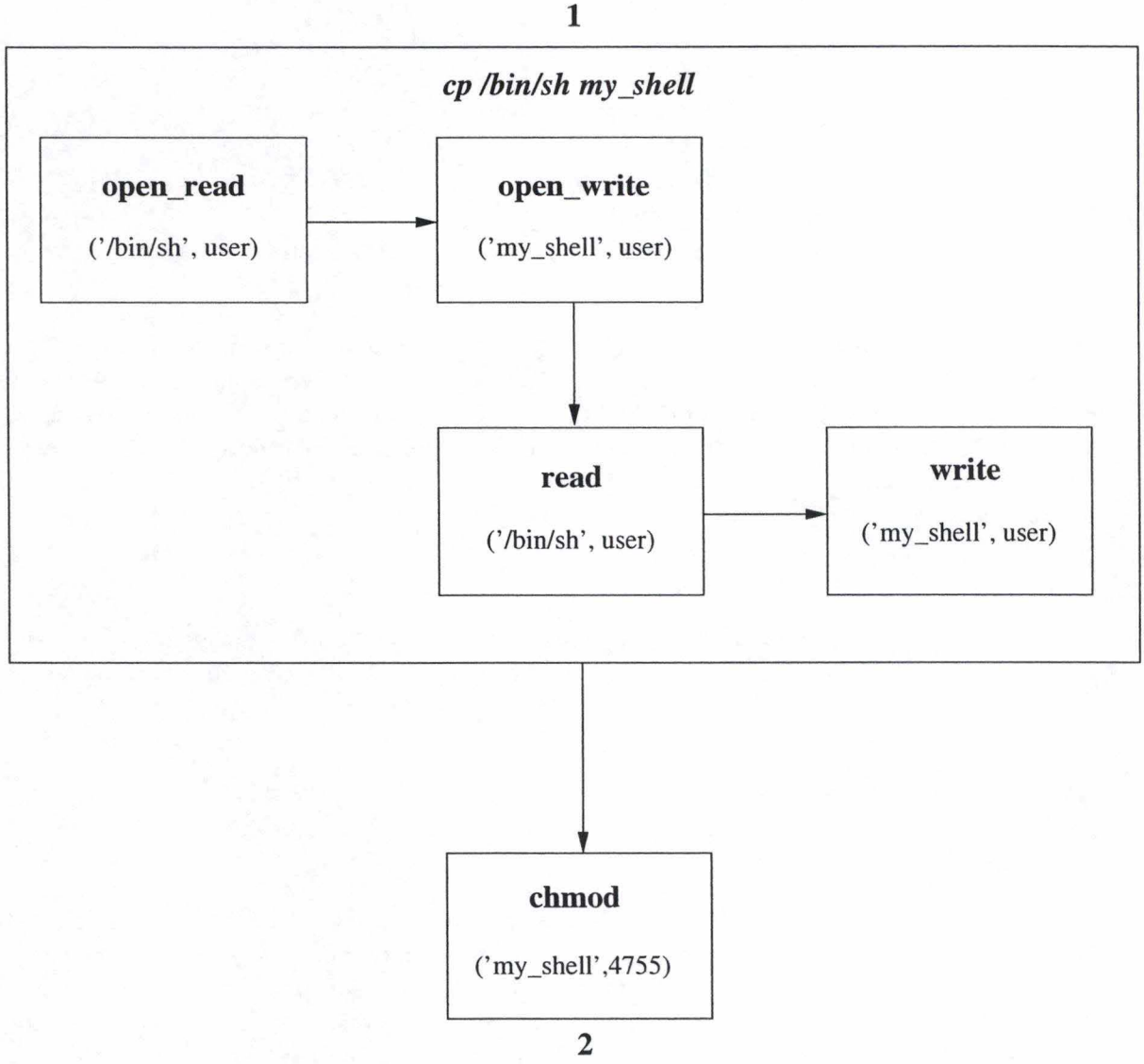


FIG. 3.6 – Détection d'un vol de shell

La détection de la copie se déroule de la même façon que la copie du fichier `/etc/passwd`. Il n'est donc pas nécessaire de le ré-expliquer. Il suffit de détecter un événement `chmod` sur la copie du shell et de constater s'il s'agit de l'application du *SetUID bit*. Dans ce cas, une alarme doit être levée. Les deux étapes de la détection sont illustrées à la figure 3.6.

Test de la détection

Les procédures de test de la validité des règles RUSSEL sont répertoriées dans le tableau suivant :

Commandes	Détection
`cp /bin/sh myshell` `chmod 4755 myshell`	OUI
`cp /bin/sh myshell` `chmod 666 myshell`	NON
`cp /tmp/toto myshell` `chmod 4755 myshell`	NON

Le code source des règles RUSSEL se trouve à la section A.2.2.

Note : Remarquons que nous ne détectons pas toutes les procédures de vol de shell différentes. En effet, l'exemple présenté ici détecte les événements générés par le programme *cp* au niveau de la copie du shell. Si l'utilisateur mal intentionné connaît le contenu des règles en cours d'utilisation, il peut dès lors contourner la détection en utilisant la séquence de commandes suivante :

```
$ touch zorglub
$ chmod 4755 zorglub
$ cat /bin/sh > zorglub
```

Cette séquence de commandes de génère pas la même suite d'événements que l'exemple précédent. Dès lors, les règles de détection que nous proposons ne peuvent détecter ce dernier cas.

Attaque sur le démon *sendmail*

Explication de l'attaque

sendmail est un démon qui, comme son nom l'indique, permet d'envoyer des emails sur le réseau Internet [SEN]. L'attaque que nous allons maintenant étudier exploite un bug présent dans une ancienne version de *sendmail*. La pré-condition pour réussir l'attaque est que le fichier `/var/mail/root` n'existe pas. L'attaque en elle-même se déroule de la manière suivante : il faut d'abord copier un shell dans un fichier nommé `root` et placer cette copie dans le répertoire `/var/mail`. Ensuite, il faut lui affecter le *SetUID bit*. A ce niveau de l'attaque, nous avons donc une copie d'un shell placée dans `/var/mail/root`. Lorsque ce shell est lancé, il s'exécute avec les droits d'accès de l'utilisateur ayant effectué la copie. Appelons-le X.

Quand un utilisateur reçoit un email, celui-ci est ajouté à la fin du fichier contenant les mails de cet utilisateur. Par exemple, si l'utilisateur Bernard reçoit un message, ce message est ajouté à la fin du fichier `/var/mail/bernard`. Le fichier contenant les emails d'un utilisateur est appelé "boîte aux lettres". Dès lors, la deuxième partie de l'attaque se déroule comme suit : en envoyant un email vide à l'administrateur (`root`), le démon *sendmail* rajoute le message à la fin du fichier `/var/mail/root`. Rappelons que ce dernier est en réalité un shell. Le message étant vide, le fichier n'est pas modifié, à ce détail près que le propriétaire

du fichier a changé : ce n'est plus l'utilisateur X qui en est le propriétaire mais bien le `root`. En effet, quand *sendmail* rajoute du courrier dans la boîte aux lettres d'un utilisateur, il vérifie également que ce fichier de boîte aux lettres appartient bien à l'utilisateur en question. Si ce n'est le cas, il réinitialise correctement l'appartenance. Cependant, et c'est là le bug de *sendmail*, il ne réinitialise pas le *SetUID bit* lors de la mise à jour du fichier d'emails. Nous avons donc maintenant un shell placé dans `/var/mail/root` appartenant au `root`, exécutable par tout le monde et ayant le *SetUID bit*. N'importe qui peut donc se faire passer pour l'administrateur et obtenir ses privilèges.

Principe de la détection

La détection de la signature se fait en quatre temps :

1. Détection de la copie du shell vers `/var/mail/root`
2. Détection de l'application du *SetUID bit* sur la copie (`chmod 4755`)
3. Détection de la création d'un fichier vide
4. Détection de l'exécution de la commande `mail` en vérifiant que le message est adressé à l'utilisateur `root` et que le corps du message est en réalité le fichier vide créé à l'étape précédente

Ces 4 étapes sont illustrées à la figure 3.7. Les trois premières étapes ne posent pas de problème. La quatrième étape est un peu plus subtile : il faut capturer l'événement correspondant à l'exécution du programme *mail* et sauvegarder son PID. Il faut ensuite rechercher l'événement correspondant à la lecture du fichier vide et vérifier le PID du processus effectuant cette lecture. Il faut enfin s'assurer que ce PID est identique au PID du programme *mail*.

Remarquons que la règle développée se base uniquement sur les quatre étapes mentionnées. Elle ne dépend donc pas de la réussite de l'attaque. Si l'utilisateur malicieux effectue sa tentative sur une version corrigée de *sendmail*, une alerte sera quand même levée. Nous pensons qu'il est malgré tout utile de tenter de détecter des attaques pour lesquelles on se sait immunisé. Cela permet de repérer les "individus douteux". On peut alors les dissuader de tenter de nouvelles attaques.

Notons que la règle de détection telle qu'elle est présentée ici ne permet pas de détecter toutes les variantes de l'attaque. Ainsi, si l'attaquant crée le fichier vide avant d'effectuer la copie du shell, l'attaque passera inaperçue. Pour la détecter, il faut écrire une variante qui tient compte de cette modification dans la séquence des étapes de l'attaque.

Ecrire n variantes d'une même règle peut rapidement devenir fastidieux. Nous présentons à la section 3.4.3 une solution permettant de faciliter l'écriture de règles possédant de nombreuses variantes.

Test de la détection

Les procédures de test de la validité des règles RUSSEL sont répertoriées dans le tableau suivant :

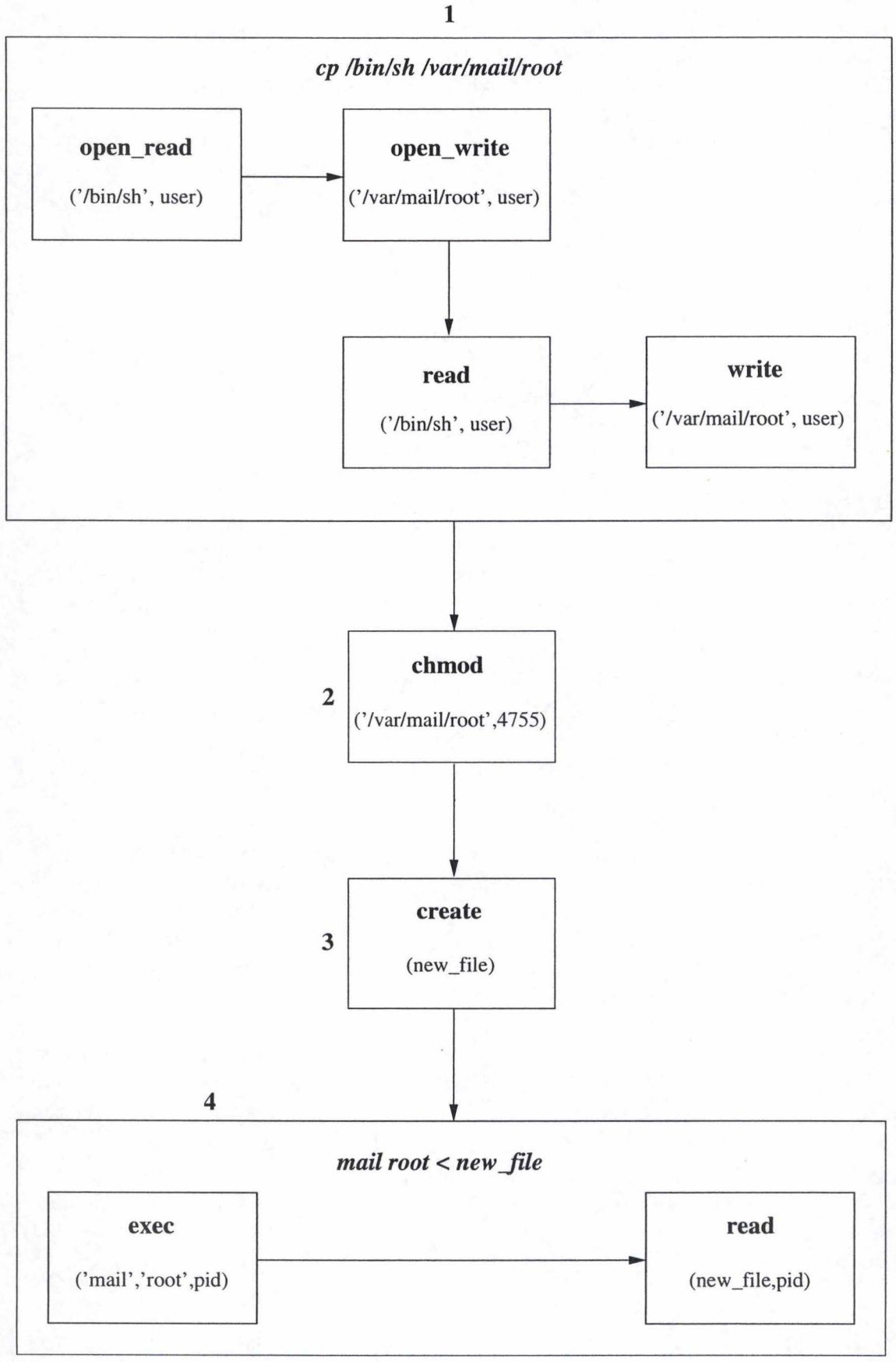


FIG. 3.7 – Détection d'une attaque sur le démon *sendmail*

Commandes	Détection

`cp /bin/sh /var/mail/root`
`chmod 4755 /var/mail/root`
`touch x`
`sendmail root < x`
	OUI

`cp toto /var/mail/root`
`chmod 4755 /var/mail/root`
`touch x`
`sendmail root < x`
	NON

`cp /bin/sh /var/mail/root`
`chmod 666 /var/mail/root`
`touch x`
`sendmail root < x`
	NON

`cp /bin/sh /var/mail/root`
`chmod 4755 /var/mail/root`
`touch x`
`sendmail zorglub < x`
	NON

Le code source des règles RUSSEL se trouve à la section A.2.3.

Détection d'une attaque "user to root" sur le programme *eject*

Explication de l'attaque

Nous avons décrit à la section 1.3.2 le principe d'une attaque *user-to-root* exploitant un *buffer overflow*. Nous présentons donc à présent une règle de détection d'une telle attaque effectuée sur le programme *eject*.

Ce programme à la propriété d'être *SetUID root*. Il s'exécute donc avec les privilèges de l'administrateur. Le code malicieux qui sera inséré dans le buffer a pour effet de lancer le programme de shell `sh`. Etant donné que pendant le temps de l'exécution, *eject* dispose des droits *root*, le shell lancé bénéficie des mêmes avantages. L'attaquant dispose donc d'un shell doté des privilèges de l'administrateur.

Principe de la détection

L'attaque se résume à un code C d'une centaine de lignes. Après l'avoir compilé, nous avons tracé son exécution grâce à notre traceur RUSSEL. Nous avons ensuite utilisé la signature obtenue pour construire une règle de détection.

La signature est composée de 3 événements à détecter :

1. Détection de l'exécution du programme *eject*
2. Détection du passage en mode administrateur (événement *set_uid*)
3. Détection de l'exécution d'un shell

Les 3 événements doivent être provoqués par un même processus. Il convient donc de s'assurer que les PID associés à ces trois événements sont identiques. Ces trois étapes sont illustrées à la figure 3.8.

Le code source des règles RUSSEL se trouve à la section A.2.4.

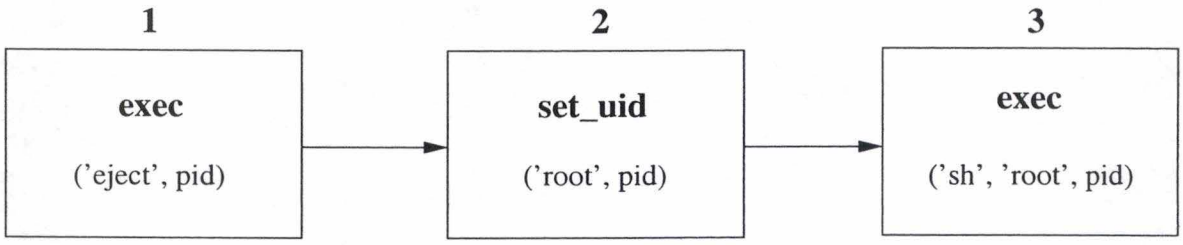


FIG. 3.8 – Détection d'une attaque *user-to-root* sur le programme *eject*

Détection de tentatives de connexion à des chevaux de Troie

Explication de l'attaque

Le principe d'une attaque utilisant un cheval de Troie a été exposée à la section 1.3.1. La plupart des chevaux de Troie n'ont pas une action destructrice immédiate ; ils se contentent d'ouvrir une brèche dans la sécurité de l'hôte attaqué en ouvrant un numéro de port prédéfini et en y attendant une connexion. Si un attaquant est au courant qu'une machine a été compromise, il peut se connecter sur le port ouvert par le cheval de Troie et lui envoyer des commandes qu'il exécutera sur le système. L'attaquant peut donc contrôler le système totalement à distance. Le lecteur attentif objectera que pour être utile à l'attaquant, le cheval de Troie doit posséder les droits *root*. Or, il n'est pas du tout certain qu'il ait été lancé par l'administrateur. En pratique, la majorité des chevaux de Troie sont destinés à être exécutés sur les systèmes d'exploitations de la famille Microsoft Windows 9X. Ce système ne disposant pas de notion de droit d'accès, les programmes, en ce compris les chevaux de Troie, ont un accès illimité aux ressources de la machine.

Principe de la détection

Il existe sur Internet de nombreux documents listant les chevaux de Troie connus ainsi que, pour chacun d'eux, le numéro de port qu'ils écoutent. Grâce à l'intégration de la source d'audit réseau *TCPlogger* dans ASAX, il a été possible d'écrire un ensemble de règles RUSSEL permettant de détecter les tentatives de connexions à ces "ports litigieux".

Le code source des règles RUSSEL se trouve à la section A.2.5.

3.4.3 Création de fonctions utilisables par le langage RUSSEL

Comme nous l'avons vu à la section 2.3.2, il est possible d'étendre le langage RUSSEL en y ajoutant des primitives implémentées en C. Cette section traite de deux applications de ce type : tout d'abord un module de traitement des adresses IP, ensuite un module de gestion de variables dynamiques.

Module de traitement des adresses IP

Une adresse IP est codée sur un entier de 32 bits. Quand l'évaluateur affiche le contenu d'une variable contenant une adresse IP, c'est donc sous la forme d'un nombre entier. Cependant, pour faciliter la lecture de tout un chacun, ces adresses sont habituellement représentées au format décimal pointé de la forme `X.X.X.X`. Nous avons donc conçu quelques fonctions permettant de passer facilement d'un format de représentation à l'autre. Les fonctions fournies sont les suivantes :

- `char* getIP(int ip)`

  Cette fonction traduit `ip`, une adresse IP codée sous forme d'entier 32bits, en une chaîne de caractères de la forme `X.X.X.X`.

- `int IPtoInt(char* ip)`

  Cette fonction traduit `ip`, une adresse IP codée sous forme d'une chaîne de caractères `X.X.X.X` en un entier 32bits.

- `int inDomain(int ip, int ip_dom, int mask)`

  Cette fonction permet de tester l'appartenance de `ip` au domaine `ip_dom` selon le masque de sous-réseau `mask`.

Le code source de ce module se trouve à la section B.1.

Intégration de variables dynamiques dans RUSSEL

Le langage RUSSEL propose deux types de variables : locales et globales. Il peut être utile de pouvoir partager de l'information entres certaines instances de règles RUSSEL. Les variables globales ne sont pas adaptées à cette situation. En effet, les variables sont dans ce cas partagées entre toutes les règles.

Illustrons notre propos avec l'algorithme 3.1. Il utilise une variable globale `g` et présente deux règles, `r1` et `r2`, respectivement paramétrées par les entiers `P1` et `P2`. Décrivons le fonctionnement de ces règles :

- La règle d'initialisation crée 2 instances des règles `r1` et `r2` avec les paramètres 10 et 20.
- La règle `r1` reçoit `P1` comme paramètre et effectue un test sur ce dernier. Si celui-ci est concluant, la variable globale g est incrémentée. La règle se redéclenche ensuite de manière à pouvoir analyser l'enregistrement suivant.
- Le fonctionnement de la règle `r2` est similaire, excepté le fait que la variable globale est décrémentée au lieu d'être incrémentée.

Dans de telles conditions, les 4 instances de règles créées par la règle d'initialisation agissent sur la même variable `g`.

Si l'on souhaite que le partage d'une variable soit restreint à certaines instances de règles, il est nécessaire de pouvoir manipuler des variables dynamiques dans le code RUSSEL. C'est-à-dire des variables que l'on peut créer et détruire à tout moment et manipulables via des pointeurs. Il nous a dès lors été demandé d'implémenter la gestion de ce type de variables. Les variables dynamiques ne constituent pas un nouveau type de données dans le langage RUSSEL. Ces variables ne sont pas utilisées directement par la machine virtuelle. Toute manipulation sur ces variables doit se faire par l'intermédiaire de fonctions auxiliaires que nous avons développées.

Le fonctionnement de notre module est simple. Les variables sont stockées dans un tableau d'entiers et sont adressables par leur position dans ce tableau. Cela rappelle bien sûr la notion de pointeur en C ; le tableau étant en quelque sorte la mémoire vive.

Voici les fonctions que nous proposons :

Algorithme 3.1 Exemple d'un code RUSSEL utilisant une variable globale g

```
1  internal global var g;
2
3  rule r1(P1 :integer);
4  begin
5          if (strToInt(header32_event) = P1)
6          -> begin
7                  g := g + 1
8          end
9          fi;
10         trigger off for_next r1(P1)
11 end;
12
13 rule r2(P2 :integer);
14 begin
15         if (strToInt(header32_event) = P2)
16         -> begin
17                 g := g - 1
18         end
19         fi;
20         trigger off for_next r2(P2)
21 end;
22
23 init_action;
24 begin
25         g := 0;
26         r1(10);
27         r2(10);
28         r1(20);
29         r2(20)
30 end.
```

- `int newvar(int val)`

  Cette fonction crée une nouvelle variable dynamique et l'instancie à la valeur `val`. La valeur de renvoi est "l'adresse" de la variable créée (sa position dans le tableau). On peut assimiler le rôle de cette fonction à la fonction `malloc()` en C.
  Exemple de création d'une variable dynamique :

```
rule r;
var x : integer;
begin
      x := newvar(0);
      ...
end;
```

- `deletevar(int var)`

  Libère la mémoire allouée par la variable `var`. Cette fonction est semblable à la fonction `free` en C.

- `setvar(int var, int val)`

  Cette fonction affecte la valeur `val` à la variable `var`.

- `getvar(int var)`

  Cette fonction renvoie la valeur de la variable `var`.

- `inc(int var)`

  Cette fonction incrémente la valeur de la variable `var`.

- `dec(int var)`

  Cette fonction décrémente la valeur de la variable `var`.

L'algorithme 3.2 illustre un exemple de l'application des variables dynamiques au sein de règles RUSSEL. Il s'agit exactement du même scénario que l'algorithme 3.1. La règle `r1` dispose d'un paramètre supplémentaire : il s'agit d'une variable dynamique `g1`. Au lieu de modifier la variable globale `g` comme dans l'exemple précédent, elle modifie le contenu de la variable référencée par `g1` (n'oublions pas que `g1` est en réalité un pointeur). Le scénario est identique dans la règle `r2` mais avec la variable `g2`. Deux variables dynamiques sont créées dans la règle `init` : `v1` et `v2`. Nous avons deux instances de chaque règle recevant soit `v1` soit `v2`. Dans la suite, nous appellerons la première instance de `r1`, `r1_A` et la deuxième, `r1_B`. Nous procéderons de même pour `r2`. Comme l'illustre la figure 3.9, `r1_A` et `r2_A` partagent la même variable dynamique adressée par `v1`. Mais en aucun cas les modifications de `v1` par l'intermédiaire de `g1` dans `r1_A` n'interfère avec les utilisations de `v2` par l'intermédiaire de `g1` au sein de `r1_B`.

Exemple d'application des variables dynamiques

Imaginons le cas d'une attaque pouvant être caractérisée soit par la séquence d'événements "A ; B ; C ; D" soit par la séquence "A ; C ; B ; D".

Les événements sont les mêmes hormis leur ordre d'apparition. Afin de détecter cette attaque, nous avons besoin de 4 règles détectant chacune un événement. Etant donné qu'il existe deux versions de l'attaque, la séquence de déclenchement des règles diffère en fonction de la variante considérée. Ce qui veut dire que le code est également légèrement différent

Algorithme 3.2 Exemple d'un code RUSSEL utilisant deux variables dynamiques : g1 et g2

```
1  rule r1(g1 : integer; P1 : integer);
2  begin
3          if (strToInt(header32_event) = P1)
4          -> begin
5          inc(g1)
6          end
7          fi;
8          trigger off for_next r1(g1,P1)
9  end;
10
11 rule r2(g2 : integer; P2 : integer);
12 begin
13         if (strToInt(header32_event) = P2)
14         -> begin
15         dec(g2);
16         end
17         fi;
18         trigger off for_next r2(g2,P2)
19 end;
20
21 init_action;
22 var v1 : integer;
23     v2 : integer;
24 begin
25         v1 := newvar(0);
26         trigger off for_next r1(v1,10);
27         trigger off for_next r2(v1,10);
28         v2 := newvar(0);
29         trigger off for_next r1(v2,20);
30         trigger off for_next r2(v2,20)
31 end.
```

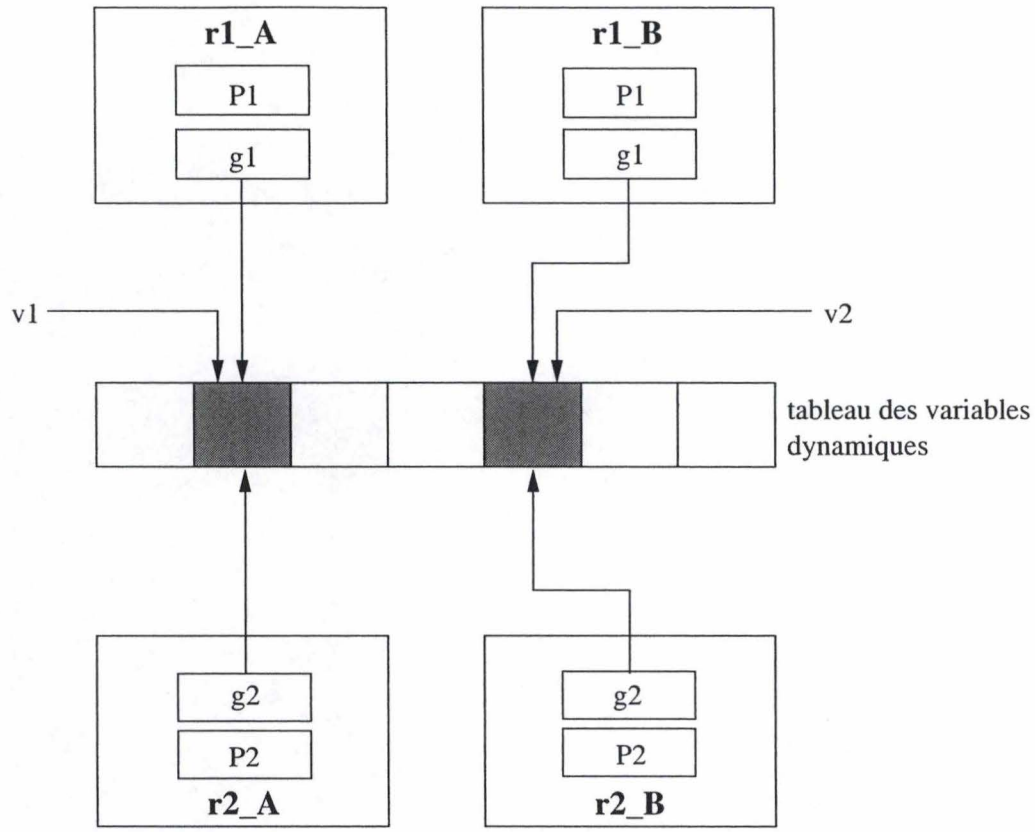


FIG. 3.9 – Distribution des variables dynamiques entre les occurrences de règles

d'une version à l'autre. Pour pouvoir détecter l'attaque de manière exhaustive, il y a lieu d'appliquer les deux versions du code des règles. Il n'est pas rare d'observer des attaques présentant cette propriété. S'il est nécessaire d'appliquer autant de versions de codes de règles qu'il y a de séquences différentes pour chaque attaque, l'administrateur risquera de vite se lasser de développer des règles identiques à quelques variations près.

Il est donc souhaitable d'imaginer un moyen de n'appliquer qu'une seule version des règles tout en gardant l'exhaustivité de la détection. Commençons par formaliser les deux séquences en une seule comme suit :

- A ; (B et C) ; D

L'idéal est de créer un ensemble unique de règles RUSSEL correspondant à cette signature. Cependant, les règles détectant une telle signature peuvent devenir très longues et rébarbatives à écrire. C'est pourquoi, l'utilisation d'un langage déclaratif de haut niveau a été clairement argumenté dans[Gér98] et [PD01]. Les travaux relatés dans ces documents nous montrent comment il est possible de générer du code RUSSEL à partir de signatures déclarées dans le langage LaDAA [Gér98] ou dans le langage Sutekh [PD01].

Notre travail sur les variables dynamiques est directement relié à Sutekh. Afin de fixer la structure des règles RUSSEL générées par ce dernier, il nous a été demandé d'étudier la question de synchronisation entre plusieurs règles RUSSEL. L'utilisation de points de synchronisation implémentés par les variables dynamiques est le fruit de cette étude. Reprenons la signature :

- A ; (B et C) ; D

Cette signature signifie qu'une fois l'événement A détecté, les détections de B et C sont lan-

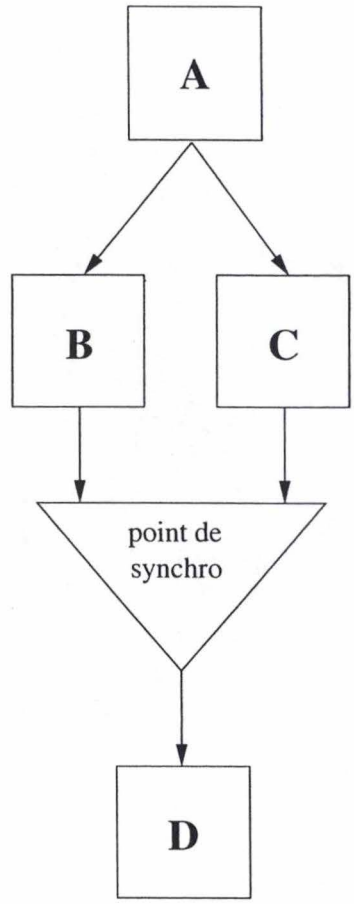


FIG. 3.10 – Modèle de la signature A ; (B et C) ; D

cées simultanément. La détection de l'événement D ne sera lancée que lorsque B et C auront été détectés, quel que soit leur ordre d'apparence. La figure 3.10 illustre ce nouveau modèle. Un point de synchronisation assure le rôle de point de rendez-vous entre les détections de B et C. Une fois qu'elles ont toutes deux effectué leur travail avec succès, la détection de l'événement D est lancée.

Afin de concrétiser cette approche en RUSSEL, nous proposons d'utiliser les variables dynamiques. Les règles détectant B et C disposent chacune d'une variable dynamique qu'elles affectent à 1 une fois qu'elles aboutissent à une détection fructueuse. Les variables sont partagées avec une autre règle (*PS*) jouant le rôle du point de synchronisation. Cette dernière vérifie continuellement les valeurs des variables dynamiques qu'elle partage. Si elles valent toutes 1, alors la règle détectant D est déclenchée. Un exemple est illustré par l'algorithme 3.3.

Ce cas d'utilisation prouve la pertinence de l'utilisation de partage de variables entre instances de règles. Dès lors, il devient possible de diminuer le nombre de règles de détection à appliquer et ainsi d'économiser le temps alloué par l'administrateur à les développer. La conception d'une règle peut ainsi rester plus proche des faits et sa gestion est simplifiée.

3.5 Mesures de performance

L'un des buts du stage était d'étudier le comportement d'ASAX distribué du point de vue de la consommation des ressources : pourcentage du temps processeur consommé,

Algorithme 3.3 Exemple d'un code RUSSEL exploitant les variables dynamiques (point de synchronisation)

```
1  rule A;
2  var sync_B;
3      sync_C;
4  begin
5          if (strToInt(header32_event) = 10)
6          -> begin
7                  sync_B := newvar(0);
8                  sync_C := newvar(0);
9                  trigger off for_next B(sync_B);
10                 trigger off for_next C(sync_C);
11                 trigger off for_next PS(sync_B, sync_C)
12         end
13         fi;
14         trigger off for_next A
15 end;
16
17 rule B(sync : integer);
18 begin
19         if (strToInt(header32_event) = 24)
20         -> begin
21                 setvar(sync,1)
22         end;
23         true -> trigger off for_next B(sync)
24         fi
25 end;
26
27 rule C(sync : integer);
28 begin
29         if (strToInt(header32_event) = 192)
30         -> begin
31                 setvar(sync,1)
32         end;
33         true -> trigger off for_next C(sync)
34         fi
35 end;
36
37 rule PS(sync_B : integer; sync_C : integer);
38 begin
39         if (getvar(sync_B) = 1) and (getvar(sync_C) = 1)
40         -> begin
41                 deletevar(sync_B);
42                 deletevar(sync_C);
43                 trigger off for_next D
44         end;
45         true -> trigger off for_next PS(sync_B, sync_C)
46         fi
47 end;
```

```
48 rule D;
49 begin
50          if (strToInt(header32_event) = 58)
51          -> begin
52                    println('Alarme!!!')
53          end;
54          true -> trigger off for_next D
55          fi
56 end;
57
58 init_action;
59 begin
60          trigger off for_next A
61 end.
```

taille mémoire occupée, ... De même, il était intéressant d'essayer de "pousser la version distribuée d'ASAX dans ses retranchements" et d'étudier sa réaction lorsque le débit des enregistrements NADF à analyser augmente. Malheureusement, nous avons passé plus de temps que prévu sur le portage et l'adaptation d'ASAX distribué. Nous n'avons donc pas pu effectuer ces mesures avec toute la rigueur qu'il se doit.

Nous avions prévu d'effectuer des mesures "à vide", c'est-à-dire sans la présence d'ASAX, sur les 25 machines impliquées dans le déploiement. Le but de ces mesures est d'obtenir un point de référence du taux d'occupation normal du processeur et de la mémoire. La période de ces mesures à vide devait s'étendre sur un mois afin qu'elles ne soient pas trop influencées par les variations saisonnières de l'utilisation des machines. Il faut en effet rappeler que ces machines sont couramment utilisées par les étudiants pour leurs divers travaux pratiques. Il ne fallait donc pas choisir une courte période de mesure sous peine de tomber dans une période de faible activité (comme les congés scolaires par exemple) ou de forte activité (comme la veille de la remise d'un rapport). Il était ensuite prévu d'installer ASAX sur ces machines et de relancer une prise de mesure à nouveau sur une période d'un mois. Nous aurions pu alors mesurer l'écart entre les moyennes de consommation avec et sans l'IDS.

Nous avons malgré tout effectué quelques mesures empiriques que nous décrivons rapidement ci-après. Si ces mesures ne permettent pas d'établir un profil exact du comportement d'ASAX, elles laissent malgré tout présager qu'ASAX n'a pas d'influence majeure sur la consommation des ressources d'une machine. En effet, sur trois jours, le taux moyen d'occupation du processeur, tous programmes confondus, est de moins de 10%. Bien que nous n'avons pas d'informations concernant les pics d'utilisation, il semble donc que l'IDS soit assez peu gourmand en puissance de calcul. Notons que ces mesures ont été effectuées alors que tous les événements système étaient enregistrés par l'audit BSM.

Nous avons également augmenté fortement le nombre d'événements générés sur une machine en lançant simultanément plusieurs instances de programmes assez "lourds" tels que *Netscape* et *Emacs*. Dans cette situation, le débit des enregistrements NADF produits a atteint trois méga-octets par minute. L'analyseur a été capable de traiter ce débit en temps réel.

Ces modestes mesures sont assez encourageantes et ne peuvent que nous inciter à refaire une prise de mesure plus rigoureuse afin d'évaluer exactement l'impact d'ASAX sur les ressources d'une machine.

3.6 Conclusion

Le stage au département informatique de l'INSA de Rennes nous a fortement éclairés sur les conditions de mise en oeuvre d'un IDS. En particulier, nous avons été positivement impressionnés par la rapidité de traitement d'ASAX ainsi que par le langage RUSSEL qui permet, en quelques minutes, d'écrire une règle de filtrage ou de corrélation entre événements.

La mise en oeuvre de la version distribuée d'ASAX a cependant mis en évidence quelques lacunes au niveau de l'implémentation et de la facilité d'utilisation. Le chapitre suivant traite des différents problèmes rencontrés et effectue une analyse critique d'ASAX distribué par rapport à ce qu'on attend d'un IDS.

Malgré tout, une fois les désagréments du portage et du déploiement surmontés, ASAX nous a confortés dans l'idée qu'il dispose d'un potentiel énorme pour la détection d'intrusion. Nous regrettons simplement de ne pas avoir pu mesurer de manière précise l'efficacité de cet IDS.

Chapitre 4

Problèmes et critique de la version actuelle d'ASAX distribué

4.1 Introduction

Ce chapitre a pour but de présenter les différentes difficultés rencontrées pendant le déploiement d'ASAX distribué effectué lors de notre stage. Nous présentons dans la section 4.2 les problèmes les plus significatifs. Les difficultés rencontrées sont dues, pour la plupart, à la forte dépendance de l'implémentation par rapport à la source d'audit BSM du système d'exploitation *Solaris 2.5*.

Le lecteur plus averti pourra ensuite consulter la section 4.3. Nous y décrivons des problèmes de moindre importance mais qui ont malgré tout rendu la tâche de déploiement plus ardue.

Nous prenons ensuite du recul par rapport aux problèmes d'implémentation pour effectuer, dans la section 4.4, une critique plus globale d'ASAX distribué. Pour cela, nous évaluerons la version actuelle d'ASAX distribué par rapport aux fonctionnalités attendues d'un IDS décrites dans [Mou97].

4.2 Problèmes significatifs rencontrés lors du portage et du déploiement

4.2.1 L'évaluateur

Mauvaise gestion des erreurs du parseur de fichier RUSSEL

Lorsqu'un évaluateur parse un code RUSSEL contenant des erreurs, il les mentionne, mais tente néanmoins une exécution de ce code. Cela est extrêmement gênant dans le cas où l'administrateur ne remarque pas les messages d'erreur dissimulés parmi les autres. Nous avouons avoir souvent été victimes de cette situation. Dans un tel contexte, les résultats des analyses deviennent tout à fait chaotiques.

Afin de résoudre ce problème, il conviendrait de modifier les évaluateurs pour qu'ils annulent l'exécution de l'analyse distribuée lorsque le code RUSSEL contient des erreurs.

```
#define NADF_TRAILER_SIZE 12
#define isTrailerRecord(p) (*(long *) (p) == NADF_TRAILER_SIZE)
```

FIG. 4.1 – Code de la macro `isTrailerRecord`

4.2.2 Le fournisseur

Risque d'erreur lors de la détection du "trailer" NADF

Les fichiers NADF se terminent par un *trailer* NADF. Il s'agit d'un enregistrement NADF bien précis destiné à marquer la fin d'un fichier NADF. La détection par le fournisseur de ce *trailer* se fait par la macro `isTrailerRecord` définie dans `fadapter.h`. Comme le montre la figure 4.1, le code de cette macro tente d'identifier un enregistrement NADF comme étant un trailer NADF en se basant uniquement sur taille de cet enregistrement.

Cela est extrêmement dangereux car tout record NADF ayant une taille de 12 octets sera considéré comme étant un *trailer record.*

A l'heure actuelle, cela ne pose pas de problème car tous les enregistrements NADF produits ont une taille supérieure à 12 octets. Mais il faut bien avouer que cette solution n'est pas très élégante. Il serait bon de réserver quelques numéros d'identifiants NADF. Par exemple, les numéros d'ID de 0 a 99 auraient une signification bien précise et pourraient être utilisés pour indiquer le début ou la fin d'un fichier NADF ou encore pour transmettre des messages entre les différents exécutables d'ASAX.

4.2.3 L'adaptateur de format

Les deux problèmes que nous allons à présent décrire sont assez techniques. Néanmoins, nous avons tenu à les présenter dans cette section car ils sont représentatifs du fait que l'implémentation d'ASAX est fort "monolithique" et qu'une modification du code se répercute à de nombreux endroits.

Mauvais calcul de la taille de l'enregistrement BSM lu

Nous avons vu à la section 2.4 que l'adaptateur de format découpait les fichiers d'audit natif et NADF en "tranches de trois méga-octets". Pour accomplir cette tâche, il doit notamment accumuler la taille des enregistrements qui ont été lus. La taille de l'enregistrement BSM qui vient d'être traduit est calculée par la fonction `getNativeRecordSize` définie dans le fichier `fadapter.c`. Dans certains cas, la taille calculée peut être incorrecte.

Cette fonction a une manière de procéder assez étrange. Pour calculer la taille d'un enregistrement BSM, elle prend en paramètre un pointeur vers l'enregistrement NADF correspondant à sa traduction. Elle se déplace ensuite dans ce dernier de 14 octets vers la droite et lit un entier de type `long`. Cet entier est considéré comme étant la taille de l'enregistrement BSM correspondant à l'enregistrement NADF traité.

Afin de comprendre cette manière de procéder, il est importe d'expliquer brièvement la structure des enregistrements BSM ainsi que le fonctionnement général de la fonction traduisant un enregistrement BSM en un enregistrement NADF.

Un enregistrement BSM représente un événement système. Chaque enregistrement est composé d'un certain nombre de *tokens*, eux-mêmes constitués de champs. Un *token* donne

Token ID	byte count	version #	event ID	ID modifier	date and time
1 octet	4 octets	1 octet	2 octets	2 octets	8 octets

FIG. 4.2 – Structure du token HEADER32 (BSM)

Taille du record NADF	ID NADF	Lg NADF	Data (ici Token ID BSM)	ID NADF	Lg NADF	Data (ici byte count)	.....
4 octets	2 octets	2 octets	1 octet + 1octet (alignement)	2 octets	2 octets	4 octets	

14 octets

FIG. 4.3 – Début de la structure d'un enregistrement NADF

des indications sur un événement : son nom (ouverture de fichier, login, ...), l'identité de la personne l'ayant effectué, le résultat (succès ou échec), ...

Il n'existe qu'un nombre limité de *tokens* et, pour chacun d'entre eux, on connaît précisément les champs qui les composent. Notons que, quel que soit le *token*, le premier champ est toujours un identifiant. Il suffit donc de lire cet identifiant pour déterminer le *token* auquel on a affaire.

Si la structure des *tokens* est stricte et bien définie, il en va autrement pour la structure des enregistrements. En effet, les *tokens* composant un enregistrement varient en fonction de la configuration de l'audit BSM. L'audit pouvant être reconfiguré à la volée, il est donc possible qu'au sein d'un même fichier d'audit, deux types d'événements identiques soient représentés par deux enregistrements ne contenant pas les mêmes types de *tokens*. Seule certitude, un enregistrement contient toujours un token de type `HEADER32` permettant d'identifier l'événement représenté par l'enregistrement. Il existe cependant une exception : le *token* `OTHER_FILE32` constitue un enregistrement à lui seul et n'est donc pas accompagné du *token* `HEADER32`. Le *token* `HEADER32`, dont la structure est illustrée par la figure 4.2, contient différents champs, dont le champ `byte count` indiquant la taille totale de l'enregistrement BSM.

Afin de traduire un enregistrement BSM au format NADF, la fonction de conversion se base sur un tableau de conversion qui, à chaque *token*, lui associe un numéro identifiant NADF. Les *tokens* sont traduits dans l'ordre de leur apparition dans cette table et non pas dans l'ordre de leur apparition au sein de l'enregistrement BSM. Donc, après traduction, l'ordre des champs dans un enregistrement NADF n'est pas forcément le même que l'ordre des tokens dans l'enregistrement BSM original. En particulier, l'ordre d'apparition des champs dans l'enregistrement NADF est lié au classement des *tokens* dans le tableau de conversion.

Dans le code d'origine de l'adaptateur de format, le *token* `HEADER32` était le premier *token* de la table. Comme un enregistrement BSM contient toujours ce *token* et que celui-ci est toujours traduit en premier lieu, on est donc certain que les enregistrements NADF auront toujours le même début de structure. En examinant la structure d'un enregistrement NADF décrite à la figure 4.3, on s'aperçoit que la traduction du champ `byte count` commencera toujours au 14 ème octet de l'enregistrement. Nous connaissons désormais la raison pour laquelle la fonction calculant la taille d'un enregistrement BSM va chercher cette information en lisant les octets 15 à 18 d'un enregistrement NADF.

Cette façon de procéder pose évidemment deux problèmes :

```
#define isOtherFileRecord(p) ((*(u_short *) p) == 0x11)
```

FIG. 4.4 – Code de la macro `isOtherFileRecord`

1. Si le token HEADER32 ne se trouve plus en début de table de conversion, le calcul est faussé.
2. Suite au passage à Solaris 2.7, le système produit un token HEADER64 au lieu d'un token HEADER32 sur les machines 64 bits. Comme nous avons rajouté ce nouveau token à la fin de la table de traduction, l'adaptateur de format lisait une donnée quelconque au lieu de la longueur de l'enregistrement BSM. Il arrivait donc que l'adaptateur considère la donnée "chemin d'accès à un exécutable" comme étant la taille de l'enregistrement BSM. Cette "taille" s'élevant à plus de 700 Mb, l'adaptateur de format demandait tout à fait normalement à l'audit BSM de changer de fichier car le seuil des 3 Mb avait été dépassé.

Pour résoudre ce problème, nous avons paré au plus pressé : nous avons simplement déplacé l'entrée correspondante au token HEADER64 en début de table de conversion.

Nous ne percevons pas pourquoi le code original se base sur un enregistrement NADF pour extraire des données concernant typiquement un enregistrement BSM. Pourquoi ne pas utiliser la fonction de conversion BSM vers NADF pour renvoyer la taille de l'enregistrement BSM venant d'être traduit ?

Mauvaise détection de la fin du fichier natif BSM

Les fichiers BSM contiennent toujours en début et en fin de fichier le *token* OTHER_FILE32. Celui-ci contient le chemin d'accès complet au fichier BSM précédent et suivant. Ainsi, lorsque l'adaptateur de format rencontre un tel *token*, il doit logiquement passer au fichier BSM suivant. Mais quand nous avons modifié la fonction de conversion BSM vers NADF pour qu'elle supporte le nouveau format BSM 2.7, cette détection ne se faisait plus correctement.

L'adaptateur teste l'identification du *token* OTHER_FILE32 non pas en examinant l'enregistrement BSM lu, mais en examinant l'enregistrement NADF produit à partir du BSM. Cette procédure est définie dans la macro `isOtherFileRecord` dans le fichier `fadapter.h`. Cette macro, illustrée à la figure 4.4, teste simplement la valeur du champ identifiant NADF (dans notre cas, il s'agissait de la valeur 0x11), ce qui amène deux problèmes :

1. Afin de prendre en compte les nouveaux tokens introduits dans BSM 2.7, nous avons dû ajouter de nouveaux identifiants NADF et modifier ceux déjà existants. Parmi les identifiants modifiés se trouvait celui utilisé lors de la conversion du *token* OTHER_FILE32. Le numéro d'identifiant étant codé en dur dans la macro, celle-ci était devenue erronée.
2. Sur Solaris 2.7, en fonction du type de processeur, certains *tokens* contiennent des informations codées sur 64 bits. Le token OTHER_FILE en fait partie. Il se peut que le système génère un *token* OTHER_FILE64 au lieu d'un *token* OTHER_FILE32. Ces *tokens* différents ont bien évidemment des identifiants différents une fois convertis en NADF.

   La macro ne gérait pas ces deux identifiants.

Nous avons modifié la macro pour que le test soit cohérent par rapport aux identifiants choisis. A terme, nous pensons cependant qu'il vaudrait mieux abandonner l'utilisation de

```
asaxd:Evaluator(0)@bonnie subject32_user_ID       : 4230
asaxd:Evaluator(0)@bonnie subject32_gro
asaxd:Evaluator(0)@bonnie up_ID      : 30185
asaxd:Evaluator(0)@bonnie
asaxd:Evaluator(0)@bonnie subject32_real_user_ID  : 4230
asaxd:Evaluator(0)@bonnie subject32_real_group_ID : 30185
asaxd:Evaluator(0)@bonnie subject32_process_ID    : 7427
```

FIG. 4.5 – Exemple de problème d'affichage dans la console

cette macro et se tourner vers une solution qui ne dépende pas des identifiants NADF utilisés. En effet, pourquoi tester une information propre à un enregistrement BSM sur un enregistrement NADF ? De plus, nous ne perçevons pas l'utilité de la traduction de ces tokens. En effet, ceux-ci ne représentent aucun événement système et ne sont donc pas utiles pour la détection d'intrusion.

4.3 Problèmes mineurs d'implémentation

4.3.1 La console

Affichage de la console

Quand un évaluateur effectue un affichage de message via l'instruction `println` du langage RUSSEL, ce message transite via le réseau jusqu'à la console pour être affiché à l'écran de l'administrateur. Malheureusement, comme le montre l'exemple de la figure 4.5, il arrive que certains messages soient coupés en deux.

Ce problème est très important au niveau de l'ergonomie de l'interface avec l'administrateur. Nous avons préféré ne pas nous y attarder pendant le stage. Néanmoins, il faudra à terme penser à améliorer la communication entre ASAX et l'administrateur. Il serait dommage que ce dernier juge ASAX inutilisable sur la seule base du fait qu'il doive reconstituer les messages lui parvenant et ce, à cause d'un bug d'affichage.

Problème de messages non affichés

Toute chaîne de caractères générée par un code C est terminée par le caractère `null` (*null-terminated string*). Les chaînes de caractères présentes dans les enregistrements BSM sont de ce type. Lors de la conversion NADF, la chaîne est recopiée telle qu'elle et est donc toujours de type *null-terminated string*.

Quand l'instruction RUSSEL `println` tente d'afficher une chaîne *null-terminated string*, par exemple l'affichage d'un chemin d'accès à un ficher (ex : `/bin/sh`), les `println` suivants au sein de la même règle ne produisent plus d'affichage. La cause de ce problème nous est inconnue. Après enquête, il s'avère que l'instruction `println` envoie la chaîne à afficher caractère par caractère sur le réseau. Comme les chaînes au format RUSSEL s'affichent impeccablement et ne sont pas terminées par *null*, nous avons choisi de supprimer tous les caractères ñull des *null-terminated string* à afficher. A présent, l'affichage est à 100% opérationnel. Cependant, nous ne sommes pas satisfaits de cette solution car nous avons appliqué une rustine, sans connaître plus avant l'origine du problème.

4.3.2 L'évaluateur

Mauvais fonctionnement de la fonction `match`

La fonction `match` est utilisée dans le cadre de "matching" d'une chaîne de caractères avec une expression régulière. Elle prend deux paramètres en entrée : le premier est l'expression régulière, le deuxième la chaîne à "matcher". Nous avons constaté quelques problèmes quand l'expression régulière est passée sous la forme d'une variable à la fonction.

Détaillons les 3 exemples illustrés par les algorithmes 4.1, 4.2 et 4.3. Nous distinguons dans la suite "variable RUSSEL", une variable déclarée dans le code RUSSEL, et "variable NADF", une variable correspondant à un champ d'un enregistrement NADF.

1. Dans le premier exemple, l'évaluateur doit afficher la chaîne `gagné` si le matching de deux constantes réussit. Ces deux constantes sont toutes les deux égales à `hello`. Le résultat de la fonction ne peut être que `vrai`. Dans ce cas, le système se comporte correctement : la chaîne `gagné` est bien affichée.
2. Dans le deuxième exemple, il s'agit de "matcher" la constante `hello` avec la variable RUSSEL `s`. Cette variable est de type `string` et a été initialisée à la valeur `hello`. Lors de l'appel à la fonction `match`, le système s'arrête brutalement sans fournir un seul message d'erreur.
3. Dans le dernier exemple, nous tentons toujours d'effectuer un matching entre la constante `hello` et la variable NADF `s`. Cette variable est toujours du type `string` mais elle contient cette fois-ci la valeur d'un champ NADF au format *null-terminated string*. Avant que tout ne se ferme, le système renvoie l'erreur :

```
Internal 344 : too much string constants : emulate() of emulate.c
```

Algorithme 4.1 Utilisation de la fonction `match` avec une constante comme expression régulière

```
if (match('hello','hello') = 1) -> println('gagné')
fi;
```

Algorithme 4.2 Utilisation de la fonction `match` avec une variable RUSSEL comme expression régulière

```
var s : string;
begin
        s := 'hello';
        if (match(s,'hello') = 1) -> println('gagné')
        fi
end;
```

La fonction `match` est très utile dans l'élaboration des règles. Il convient donc de régler rapidement ce problème ; le temps nous a manqué pour le faire.

Les entiers en RUSSEL sont systématiquement signés

La fonction `strToInt` renvoie un entier signé codé sur 4 octets. Cela a entraîné quelques problèmes quand nous avons voulu traiter les données en provenance de la source réseau (*TCPlogger*).

Algorithme 4.3 Utilisation de la fonction `match` avec une variable NADF contenant une chaîne de caractères de type *null-terminated string* comme expression régulière

```
var s : string;
begin
        s := path_0_path;
        if (match(s,'hello') = 1) -> println('gagné')
        fi
end.
```

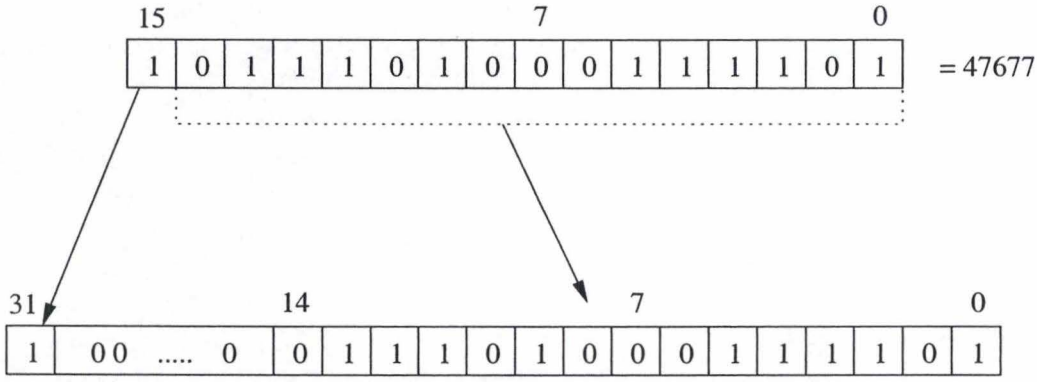


FIG. 4.6 – Problème de signe lors de la conversion d'un numéro de port 16bits au format 32bits

En effet, dans le protocole TCP, les numéros de ports sont codés sur deux octets et sont interprétés comme étant non-signés. Quand la fonction `strToInt` traite un champ NADF contenant la valeur d'un numéro de port, elle considère que ce numéro de port est un entier codé sur 2 octets, mais signé.

Le résultat renvoyé par la fonction est donc faux puisque les 15 bits de poids faibles de la donnée d'origine sont recopiés à la fin de l'entier RUSSEL et le bit de poids fort (le bit de signe) est recopié au début du bloc de 4 octets. Or ce bit de poids fort n'est pas un bit de signe puisque la donnée d'origine est non-signée. Comme l'indique la figure 4.6, cela a pour effet que tous les numéros de port supérieurs à 32768 sont considérés comme étant négatifs (tous les numéros ayant le bit de poids fort à 1).

Pour contourner cette limitation, une solution possible est de créer une autre fonction `strToInt` prenant en compte le fait que les données d'origine sont non-signées. Nous avons adopté une autre solution : l'adaptateur de format *TCPlogger* a été modifié pour que les champs NADF représentant des numéros de ports contiennent en réalité des entiers signés codés sur 4 octets. Cette modification est assez simple. Il suffit de préfixer la valeur du numéro de port par 16 bits mis à zéro avant de l'enregistrer dans un champ NADF.

Arrêt brutal de l'évaluateur lorsque le fichier RUSSEL spécifié n'est pas présent

L'évaluateur s'arrête brutalement via un appel à `exit(1)` quand le fichier RUSSEL spécifié n'est pas présent. Aucun message de terminaison n'ayant été envoyé par cet évaluateur à la console ou à l'évaluateur maître, ces derniers ne sont pas informés de cette terminaison abrupte. Ce qui peut entraîner l'existence d'évaluateurs maîtres attendant des messages d'un esclave n'existant plus.

Pour remédier à ce problème nous avons propagé l'information de terminaison vers la console et l'évaluateur maître.

4.3.3 Le fournisseur

Quand nous avons exploré le code source du fournisseur, nous nous sommes rendu compte qu'un grand nombre de fonctions, lorsqu'elles rencontraient un problème, arrêtaient brutalement le processus via l'instruction d'arrêt `exit(1)`. Cet arrêt brutal entraînait un problème particulièrement gênant : lorsqu'un ordre d'arrêt de l'évaluation distribuée est envoyé depuis la console, il est d'abord acheminé aux fournisseurs qui le relayent ensuite à leur évaluateur respectif. Le fournisseur s'étant arrêté "brutalement", le relais du message d'arrêt n'est plus effectué. Il devient donc impossible d'arrêter les différents évaluateurs depuis la console de l'administrateur. La seule solution est donc de les arrêter "à la main". Ceci est acceptable si ASAX s'exécute sur un seul hôte, mais si celui-ci est déployé sur 25 hôtes, cela veut dire qu'il faut réitérer 25 fois la séquence :

1. connexion distante sur la machine
2. élimination des processus orphelins
3. déconnexion

Si des erreurs apparaissent toutes les heures, l'administrateur risque de très vite se lasser et d'abandonner l'utilisation d'ASAX.

Nous avons donc remédié à ce problème de la façon suivante : lorsqu'une erreur est détectée, le fournisseur envoie un ordre d'arrêt à son évaluateur puis seulement termine son exécution. Malheureusement, il reste encore quelques cas non corrigés car ces derniers demandent plus d'investissement quand à la modification du code.

4.3.4 L'adaptateur de format

Dérèglement des ordres de changement de fichier d'audit envoyés au processus d'audit BSM

L'adaptateur de format propose une gestion de la taille maximale des fichiers BSM et NADF. Le pseudo-algorithme (cf. figure 4.7) décrivant cette gestion est relativement simple. A chaque enregistrement BSM lu, sa taille est accumulée dans un compteur. De même, chaque fois qu'un enregistrement NADF est écrit, sa taille est aussi accumulée dans un compteur. Par défaut, lorsque le fichier NADF produit atteint trois méga-octets, l'adaptateur ferme ce fichier et continue à produire du NADF dans un autre.

De même, lorsque le volume des données lues dans le fichier BSM atteint trois méga-octets, l'adaptateur envoie un signal au processus d'audit BSM lui demandant d'arrêter d'enregistrer les événements audités dans le fichier en cours et de passer à un autre fichier. Un dérèglement dans l'envoi de ce signal peut se produire dans la situation suivante :

1. ASAX envoie au processus d'audit BSM une requête de changement de fichier.
2. Entre cet envoi de requête et sa réception par le processus d'audit BSM, de nouveaux événements sont produits et enregistrés par le système.
3. Le processus d'audit BSM reçoit la requête : il rajoute le token OTHER_FILE32 à la fin du fichier BSM en cours et passe à un autre fichier.

```
lire un record BSM
si le record lu = OTHER_FILE32
   alors - passer au fichier BSM suivant
         - remise a zéro de la taille_totale_fichier_BSM

   sinon - taille_totale_fichier_BSM =
               taille_totale_fichier_BSM + taille du dernier record BSM lu

convertir le record BSM en record NADF
écrire le record NADF

taille_totale_fichier_NADF =
         taille_totale_fichier_NADF + taille du dernier record NADF produit

si taille_totale_fichier_NADF  > 3 méga-octets
   alors changer de fichier NADF

si taille_totale_fichier_BSM > 3 méga-octets
   alors demander à l'audit de passer à un autre fichier BSM
```

FIG. 4.7 – Pseudo-algorithme de l'adaptateur de format

4. ASAX lit l'enregistrement natif BSM suivant. Cet enregistrement n'est pas le *token* OTHER_FILE32, mais bien un enregistrement représentant un des événements qui ont été audités entre l'émission de la requête de changement de fichier natif et sa réception par le processus d'audit BSM.

   Cet enregistrement n'étant pas le *token* OTHER_FILE32, sa taille est ajoutée à la taille totale du fichier natif. Celle-ci dépasse dès lors les 3 méga-octets et une nouvelle demande de changement de fichier est envoyée au processus d'audit BSM. Ce cycle continue jusqu'à la lecture de l'enregistrement contenant le *token* OTHER_FILE32.

5. En découvrant le *token* OTHER_FILE32, l'adaptateur de format en extrait le nom du fichier BSM suivant et essaye de l'ouvrir. Seulement, le processus d'audit BSM ayant déjà reçu plusieurs demandes de changement de fichier et donc ayant changé ce dernier plusieurs fois de suite, il y a de fortes chances pour que le fichier "suivant" ne soit plus le fichier que la source d'audit utilise à ce moment. Or, quand celle-ci ferme un fichier pour passer à un autre, elle renomme l'ancien (en le suffixant par l'heure de fermeture). Ainsi, le nom de fichier extrait du *token* OTHER_FILE32 n'est plus valide et l'adaptateur de format est incapable de poursuivre sa conversion.

Nous avons appliqué la solution suivante : nous avons rajouté une valeur booléenne vérifiée par une condition lors de la tentative d'envoi d'un ordre de changement de fichier. Le comportement sera le suivant selon la valeur de la variable :

- **vrai** : un message a déjà été envoyé mais l'adaptateur de format n'a toujours pas sauté sur le fichier suivant. Un second message ne peut donc être envoyé sous peine d'ordonner à l'audit BSM de changer de fichier prématurément. La valeur devient "faux" une fois que l'adaptateur passe au fichier suivant.
- **faux** : un ordre de changement est envoyé au processus d'audit. La valeur devient "vrai".

Fermeture incorrecte des fichiers NADF lorsque l'adaptateur de format est terminé par la commande clavier CTRL-C ou via la commande `kill`

Lorsque l'adaptateur de format a terminé d'écrire dans un fichier NADF, il rajoute à la fin de celui-ci un *trailer* NADF. Ce *trailer* NADF est considéré par le fournisseur comme une marque de fin de fichier. Si le fournisseur arrive au bout du fichier sans trouver le *trailer* en question, il considère que le fichier NADF est un fichier *on-line* en cours de traduction. Le fournisseur attend donc indéfiniment l'arrivée de nouvelles données dans ce fichier mais, en vain, puisque la traduction a été interrompue.

Nous avons modifié le code afin que l'adaptateur de format puisse intercepter les signaux de terminaison qui lui sont envoyés et, lorsqu'il en capte un, ferme correctement le fichier NADF en cours d'écriture en lui administrant le fameux *trailer*.

4.4 Analyse critique de la version existante

Les sections 4.2 et 4.3 ont mis en évidence les différentes difficultés rencontrées lors du portage d'ASAX distribué. Cependant, juger la qualité de l'IDS sur la seule base de ces difficultés serait faire preuve d'un manque de rigueur et d'objectivité. Nous allons donc dans cette section faire abstraction de ces problèmes et tenter de dégager les qualités et les défauts intrinsèques d'ASAX distribué.

Pour cela, nous allons passer en revue les fonctionnalités souhaitées d'un système de détection d'intrusion multi-hôtes décrites dans [Mou97] et, pour chaque fonctionnalité attendue, la confronter avec la version actuelle d'ASAX. Si certaines des fonctionnalités ne sont pas satisfaites, nous vérifierons si cela est dû à une mauvaise conception de l'architecture ou à un problème d'implémentation.

4.4.1 Administration en un lieu unique

Description de la fonctionnalité attendue

L'IDS doit présenter l'ensemble des rapports de sécurité sous une seule vue globale. Il est en effet impensable de demander au responsable sécurité d'examiner des rapports situés sur des terminaux différents, qu'ils soient physiques ou virtuels. De même, il doit pouvoir intervenir sur une machine sans devoir se déplacer ou sans établir manuellement de connexion avec celle-ci. L'IDS doit donc offrir un véritable "poste de commande et d'observation" au responsable sécurité.

Description de la fonctionnalité réelle

ASAX distribué propose une console à partir de laquelle on peut observer le résultat de l'analyse des différents hôtes. Elle permet d'agir sur l'analyse en cours ainsi que de configurer certains paramètres propres à la génération de l'audit natif.

Il faut cependant souligner que, dans son implémentation actuelle, la console n'est pas des plus conviviales et souffre de quelques lacunes. Ainsi, lors de la terminaison du programme de la console, l'analyse distribuée continue de s'exécuter bien qu'aucune information la concernant ne soit sauvegardée. Ainsi, une fois la console quittée, il est impossible d'en relancer une nouvelle pour reprendre le contrôle sur le processus d'analyse. Il faut donc avant de la quitter s'assurer que l'on a bien donné l'ordre d'arrêter l'analyse distribuée.

4.4.2 Configuration flexible

Description de la fonctionnalité attendue

Il est rare de trouver un parc de machines homogènes demandant toutes le même niveau de sécurité. L'hétérogénéité des machines et de leur usage entraîne des besoins d'analyse différents. Chaque système d'exploitation possède ses failles spécifiques. Par exemple, un serveur public de fichiers ne demande pas la même surveillance qu'une simple station de travail. Il doit donc être possible d'effectuer des analyses différentes en fonction de la machine concernée. De plus, le responsable sécurité doit pouvoir modifier "à la volée" l'analyse effectuée sur une machine donnée.

Description de la fonctionnalité réelle

Le processus d'analyse d'ASAX répond pleinement à cette exigence : chaque hôte peut effectuer une analyse différente et il est possible, depuis la console, de modifier à la volée l'analyse effectuée sur un hôte.

4.4.3 Adaptabilité

Description de la fonctionnalité attendue

Le système de détection d'intrusion doit pouvoir s'adapter en fonction de la taille et de la structure de l'environnement à surveiller. Typiquement, dans les réseaux de grande taille, l'analyse doit pouvoir se faire selon une structure hiérarchique.

Description de la fonctionnalité réelle

L'architecture et l'implémentation d'ASAX distribué ne prévoient qu'une relation "maître-esclave" à un seul niveau. De plus, une fois qu'une analyse distribuée est lancée, il est impossible de rajouter dynamiquement un hôte dans le système existant. Implémenter une structure hiérarchique à plusieurs niveaux demanderait une réécriture complète de la console et des protocoles de communications utilisés entre les composants. Paradoxalement, l'évaluateur ne demande que peu de modifications pour être intégré dans une structure multi-niveaux. En effet, le même exécutable est utilisé tant pour un évaluateur maître que pour un évaluateur esclave. Rajouter des niveaux supplémentaires dans la relation "maître-esclave" ne devrait entraîner que peu, voire aucune modification à l'évaluateur.

4.4.4 Tolérance aux pannes

Description de la fonctionnalité attendue

Dans un environnement où la sécurité est critique, l'analyse doit s'effectuer de manière ininterrompue. Une panne d'une des machines doit avoir un impact minimal. L'analyse doit se poursuivre de manière transparente pour les autres machines. Si une défaillance de la machine maître survient dans une architecture "maître-esclaves", le responsable sécurité doit pouvoir désigner une autre machine maître afin qu'elle reprenne l'analyse là où elle s'était arrêtée.

Description de la fonctionnalité réelle

Dans la version distribuée d'ASAX, l'arrêt d'un hôte esclave ne perturbe pas le comportement des autres analyseurs. Une panne de l'hôte maître est, quant à elle, fatale pour le système ASAX.

4.4.5 Portabilité

Description de la fonctionnalité attendue

Puisque l'on se trouvera le plus souvent face à un ensemble d'hôtes hétérogènes, le portage du système IDS doit être réduit à sa plus simple expression. Il importe donc que les parties de code dépendantes de la plate-forme soient correctement identifiées et isolées.

Description de la fonctionnalité réelle

D'un point de vue conceptuel, l'utilisation du format de données NADF permet une indépendance par rapport à la source d'audit. Cependant, l'implémentation actuelle d'ASAX distribué ne facilite pas le portage. Hormis le noyau, les composants de l'IDS sont fortement dépendants de la plate-forme utilisée.

4.4.6 Sécurité

Description de la fonctionnalité attendue

Les données d'audit et les rapports générés contiennent des données sensibles. Il faut donc éviter qu'ils soient lus ou modifiés par des personnes non-autorisées. Dans le même esprit, les communications réseau entre les différents composants du système de détection d'intrusion doivent être sécurisées. Enfin, l'IDS s'exécutant souvent avec des privilèges supérieurs, il faut s'assurer qu'il ne comporte pas de faille exploitable (par exemple une attaque en *buffer overflow*).

Description de la fonctionnalité réelle

Comme décrit dans [Mou97], la sécurité n'a pas été prise en compte lors de la conception de la version distribuée d'ASAX. Les fichiers d'audit sont simplement protégés par les mécanismes de droit d'accès Unix. Les communications entre composants utilisent des *sockets* standards non sécurisés.

4.4.7 Réutilisabilité

Description de la fonctionnalité attendue

En plus des fonctionnalités attendues, il faut souligner, dans le cas d'ASAX, l'importance de l'inter-opérabilité. En effet, ASAX est à la base un analyseur de flux séquentiel dont les possibilités dépassent largement le cadre de la détection d'intrusion distribuée. Il faut dès lors s'assurer de la capacité de réutiliser des composants dans le cadre d'applications telles que l'analyse d'un flux réseau ou l'analyse de fichiers séquentiels n'ayant rien à voir avec la sécurité.

Description de la fonctionnalité réelle

Il apparaît clairement que toute l'architecture d'ASAX distribué a été pensée afin de pouvoir réutiliser le concept d'analyse de fichier séquentiels dans d'autres domaines d'application que la détection d'intrusion.

Cependant, seul le coeur d'ASAX, à savoir l'évaluateur, est réutilisable. Les autres composants sont beaucoup trop liés au domaine de la détection d'intrusion et certaines optimisations mises en oeuvre font qu'il est beaucoup plus facile de réécrire complètement ces composants que de chercher à les adapter.

Néanmoins, ASAX s'orientant principalement vers la détection d'intrusion, nous comprenons fort bien que cela s'en ressente au niveau de l'implémentation. Nous regrettons toutefois que le code source n'ait pas été écrit de manière à faciliter son adaptation.

4.5 Conclusion

Ce chapitre a souligné les problèmes que nous avons rencontrés lors du portage et de l'utilisation d'ASAX distribué. Nous avons vus que la plupart des faiblesses de cet IDS ne sont pas dues à une mauvaise conception de l'architecture mais bien à un manque de flexibilité au niveau de l'implémentation des composants.

Nous sommes conscients que certaines des solutions que nous avons apportées ne font que masquer temporairement ces problèmes. Si un portage devait à nouveau être réalisé, ceux-ci ressurgiraient certainement. Ceci dit, une modification en profondeur de l'implémentation d'ASAX distribué est une tâche qui doit être longuement réfléchie et préparée avec soin.

Nous allons donc dans la suite de ce mémoire entamer ce travail de préparation en présentant quelques pistes de réflexions afin de permettre à ASAX de se rapprocher de l'"IDS idéal", tant au point de vue de ses fonctionnalités que de sa convivialité d'utilisation.

Chapitre 5

Proposition d'une nouvelle architecture d'ASAX distribué

5.1 Introduction

D'un point de vue théorique, nous avons vu que l'architecture d'ASAX distribué convient à la majorité des cas d'utilisation. De même, par l'utilisation du format de données NADF, un portage ne devrait se résumer qu'à la modification de la fonction de traduction "format natif vers format NADF" présente au sein de l'adaptateur de format.

Cependant, le chapitre précédent a clairement montré que l'implémentation de la version distribuée d'ASAX souffre de certaines lacunes qui peuvent malheureusement le rendre moins attrayant aux yeux des utilisateurs. En particulier, notre stage a permis de mettre en évidence, un certain manque de souplesse à deux niveaux :

1. Si ASAX est effectivement bien découpé en modules, le code source de ces différents modules manque par contre de souplesse. Un portage ou une adaptation demandent un travail conséquent et, souvent, une modification du code se répercute à de nombreux endroits. Le code devrait donc être moins monolithique, plus modulaire afin de faciliter son adaptation.
2. Nous avons constaté un certain manque de souplesse au niveau de l'utilisation d'ASAX distribué. Par exemple, il est impossible de sortir du mode de fonctionnement : "L'adaptateur de format écrit le résultat de sa conversion dans des fichiers. Le fournisseur sélectionne ces fichiers sur base d'un *timestamp*, lit les enregistrements NADF et les transmet à l'évaluateur.". Dans le cadre d'une analyse exclusivement à la volée, nous aurions aimé pouvoir établir une connexion directe entre l'adaptateur de format et l'évaluateur, sans passer par un stockage de fichiers intermédiaires ou par le fournisseur.

Pour séduire les utilisateurs, il est donc souhaitable qu'ASAX quitte progressivement le statut de prototype pour se transformer en un IDS "clef en main". Nous entendons par là un système qui, après acquisition, est fonctionnel quasi sur-le-champ. Pour atteindre ce but, il est nécessaire de proposer en standard un ensemble d'outils d'aide à la configuration et à l'installation ainsi qu'une base de règles de détection importante.

Bien entendu, ces outils ne peuvent être développés que si l'implémentation d'ASAX distribué a été revue afin d'offrir plus de flexibilité. Nous allons donc entamer le travail de réflexion relatif à la révision de l'implémentation en précisant dans la section 5.2 les fonctionnalités pratiques que nous attendons de chaque module ainsi que l'organisation générale

de leur code source. Nous en profiterons pour modifier légèrement l'architecture actuelle en introduisant un nouveau composant : le *démon*. Son fonctionnement et ses implications sur l'architecture d'ASAX seront décrits en détail dans la section 5.3. Enfin, afin de prendre en compte ces quelques changements, les commandes et fonctionnalités de la console sont reprécisées et étendues dans la section 5.4.

5.2 Souplesse et confort d'utilisation : expression des besoins

Cette section présente, composant par composant, les différentes limitations que nous avons rencontrées lors de l'utilisation de la version distribuée d'ASAX. Il ne s'agit pas ici à proprement parler de problèmes entraînant un mauvais fonctionnement, mais bien de restrictions qui rendent l'utilisation du logiciel peu attirante ou fastidieuse.

Pour chaque limitation exposée, nous proposons une solution permettant d'obtenir plus de souplesse d'utilisation. La plupart de ces propositions ne modifient pas l'architecture générale mais uniquement l'implémentation.

Bien que les solutions proposées augmentent la flexibilité d'ASAX, nous sommes conscients que l'adaptabilité parfaite n'est qu'un mythe. Ainsi, il existe à coup sûr certains cas d'utilisation qui ne pourront être satisfaits par l'architecture proposée et qui nécessiteront une modification du code source. Pour faciliter l'adaptation à ces cas particuliers, on veillera à soigner l'implémentation des composants. Le code source sera toujours clair et commenté. Il devra être écrit de manière modulaire et on tentera d'isoler au maximum les parties dépendantes de la plate-forme d'utilisation. Pour les composants dont le code actuel n'est pas facilement modifiable, nous nous efforcerons de proposer une organisation du code permettant une adaptation plus aisée.

5.2.1 L'évaluateur

Transparence de l'intégration de routines C dans le langage RUSSEL

Comme nous l'avons vu à la section 2.3.2, il est possible d'intégrer des routines développées en C au langage RUSSEL. Cela offre à l'administrateur une énorme flexibilité au niveau de la personnalisation du langage RUSSEL et lui permet dès lors de concevoir des règles de détections ayant un comportement spécifique à ses besoins. La démarche à suivre pour effectuer une "mise-à-jour" de RUSSEL est expliquée dans [Mou97, MLCM93] mais peut être résumée comme suit :

1. Création et compilation des fonctions personnelles
2. Création d'un fichier descripteur des fonctions (noms, types des paramètres, type de la valeur de retour) en utilisant l'application `libman` fournie avec l'archive de la version distribuée d'ASAX.
3. Compilation d'une nouvelle version de l'évaluateur avec les nouvelles fonctions intégrées grâce à l'utilitaire `masax`, ce dernier utilisant le fichier créé au point précédent et également fourni avec l'archive

Force est de constater que cette méthode n'offre pas la meilleure flexibilité pour l'administrateur. Ce dernier, souffrant généralement d'un emploi du temps assez chargé, n'a pas vraiment envie de retenir les différentes démarches à suivre et un format supplémentaire de fichier à respecter (le fichier de description des routines C). S'il est obligé d'utiliser une

telle méthode et de la répéter sur chaque hôte à chaque "mise-à-jour", il risque très vite de trouver l'utilisation d'ASAX distribué ennuyeuse et de le laisser tomber sur base de ce seul jugement.

Ce jugement nous a été explicitement communiqué lors d'un compte-rendu de l'utilisation d'ASAX distribué à l'INSA six mois après notre départ. L'administrateur souhaite disposer d'une version plus "clef en main" de cette fonctionnalité, c'est-à-dire, juste avoir à dire "ajouter ce module" sans intervention supplémentaire.

Nous proposons dès lors l'utilisation de librairies dynamiques. Une librairie dynamique est un code objet n'ayant pas été lié avec l'exécutable principal. Le principe de l'utilisation d'une telle librairie est le chargement dynamique en mémoire des fonctions qu'elle contient. Ces dernières ne sont donc chargées que lorsque le programmeur le souhaite contrairement à l'utilisation de librairies statiques ou d'un exécutable résultant d'un "linkage" de tous les codes objets produits.

Cependant, une économie de mémoire n'est pas le but recherché ici. Notre objectif est d'étendre l'évaluateur sans pour autant le recompiler via une liste rébarbative de tâches à effectuer. Les librairies dynamiques remplissent également cet objectif. Supposons qu'ASAX dispose d'un répertoire dédié où stocker ses librairies, il peut dès lors analyser le contenu des librairies présentes dans ce répertoire à chaque lancement de son exécution[1] et donc "se mettre au courant" des fonctions annexées par l'administrateur à RUSSEL.

Grâce à l'utilisation des librairies dynamiques, l'administrateur dispose d'un gain de temps énorme. En effet, l'ajout de fonctions personnelles se fait de manière beaucoup plus transparente. Une fois qu'il a créé sa librairie dynamique, l'administrateur peut la distribuer et la copier dans chacun des répertoires dédiés aux librairies de chaque hôte faisant partie de l'IDS distribué. La nouvelle procédure d'installation de routines dans RUSSEL se résumerait dès lors à ceci :

1. Création des fonctions et compilation sous forme d'une librairie dynamique.
2. Placement de la librairie dans le répertoire dédié d'ASAX.

Remarquons que pour enlever certaines fonctions à RUSSEL ayant été ajoutées au préalable, il est de nouveau nécessaire d'utiliser le couple `libman` et `masax`. Grâce aux librairies dynamiques, il suffit juste d'effacer du répertoire la librairie contenant les fonctions que l'on souhaite supprimer de RUSSEL.

Prédisposition à l'implémentation d'un nouvel évaluateur

Si nous comparons les versions monopostes et distribuées de l'évaluateur, nous constatons que les seules différences se situent au niveau de l'"emballage". En effet, le noyau d'ASAX, ce qui constitue la machine virtuelle RUSSEL, n'a pas changé. La version distribuée diffère de la version monoposte par toutes ses fonctionnalités de gestion réseau.

Si nous dissocions l'évaluateur, nous avons donc d'un côté ce qu'on peut appeler le noyau et de l'autre les fonctionnalités propres au contexte d'utilisation. Il n'est pas improbable que l'administrateur souhaite réimplémenter l'évaluateur afin de l'optimiser en fonction du contexte et de son propre environnement d'utilisation (une modification de la gestion du réseau par exemple). Dans ce cas, il ne ferait que modifier l'"emballage". Le noyau, quant à lui, peut également faire l'objet d'une ré-implémentation via une mise à jour par exemple. Toutefois, ce rôle est attribué à l'équipe de développement d'ASAX.

[1] L'implémentation des librairies dynamiques le permet

Nous proposons de dissocier l'évaluateur physiquement : d'un côté le noyau matérialisé sous forme d'une librairie dynamique (cfr . 5.2.1) et de l'autre, l'exécutable de l'évaluateur implémenté selon le contexte d'utilisation. L'utilisateur, lors de son implémentation, ferait appel au noyau grâce à des primitives prédéfinies et donc à respecter. Ces primitives seraient regroupées dans un fichier *header* propre au langage C et seraient importées dans le code de l'évaluateur via la commande C suivante :

```
#include <asax_kernel.h>
```

Nous voyons, grâce à cette dissociation, que le code de l'évaluateur deviendrait beaucoup plus lisible et plus flexible dans le cadre d'une modification (le code de l'évaluateur monoposte, hors noyau, se résumerait à quelques dizaines de lignes). De plus, l'administrateur ne se soucierait pas du fonctionnement du noyau durant son travail de ré-implémentation.

Soulignons l'avantage supplémentaire de la possibilité de mise à jour dynamique du noyau. Imaginons le cas où l'équipe de développement d'ASAX produit un nouveau noyau plus performant. Cette nouvelle version serait d'abord distribuée aux administrateurs utilisant ASAX. Ensuite, par le simple fait d'écraser l'ancien noyau par le nouveau (rappelons qu'il s'agit d'une librairie dynamique), l'évaluateur serait mis à jour, et ce, sans procédure d'installation ou de recompilation quelconque. Nous pouvons dès lors imaginer sans peine le gain en temps et en transparence que cette possibilité offrirait dans un IDS composé de plusieurs dizaines d'hôtes.

5.2.2 L'adaptateur de format

Mode de sortie de l'adaptateur de format

Nous avons vu à la section 3.3.3 que nous avons développé un adaptateur de format *TCPlogger*. Suite à une analyse de l'évolution de l'espace disque alloué par les fichiers NADF, il a été décidé que les données NADF résultant de cet adaptateur ne seraient pas stockées mais directement envoyées à l'évaluateur. Nous avons dès lors rencontré un problème au niveau du fournisseur. En effet, l'architecture oblige l'utilisation de ce composant. Cependant, dans le cas d'une analyse *on-line* sans stockage de fichiers, le fournisseur devient inutile. Son rôle de filtre sur base du temps devient également dénué de sens puisque l'analyse se fait en temps réel. C'est pourquoi, nous avons dû déléguer le rôle de fournisseur à l'adaptateur de format, c'est-à-dire que c'est ce dernier qui se connecte directement à l'évaluateur et non plus par l'intermédiaire du fournisseur.

En conclusion, si l'on désire faire de l'analyse en temps réel, quelles que soient les sources d'audit, le fournisseur devient obsolète. C'est pourquoi nous proposons de prévoir, dans l'implémentation de tout adaptateur de format, la possibilité de choisir le mode de sortie de la conversion NADF, c'est à dire, comme l'illustre la figure 5.1 : l'écriture dans un ou des fichiers, ou bien l'envoi direct à l'évaluateur. Cette possibilité offrirait plus de flexibilité au niveau de l'organisation de l'IDS par l'administrateur selon les ressources dont il dispose.

5.2.3 La console

Terminaison de la console

La console lance à distance les exécutables sur les différentes machines du réseau. Ce composant a donc connaissance de toute la structure du réseau d'évaluateurs. Elle connaît quelles sont les machines faisant partie de l'analyse distribuée, le nombre d'évaluateurs qu'elles hébergent, le fichier de règles RUSSEL analysé par chaque évaluateur, ...

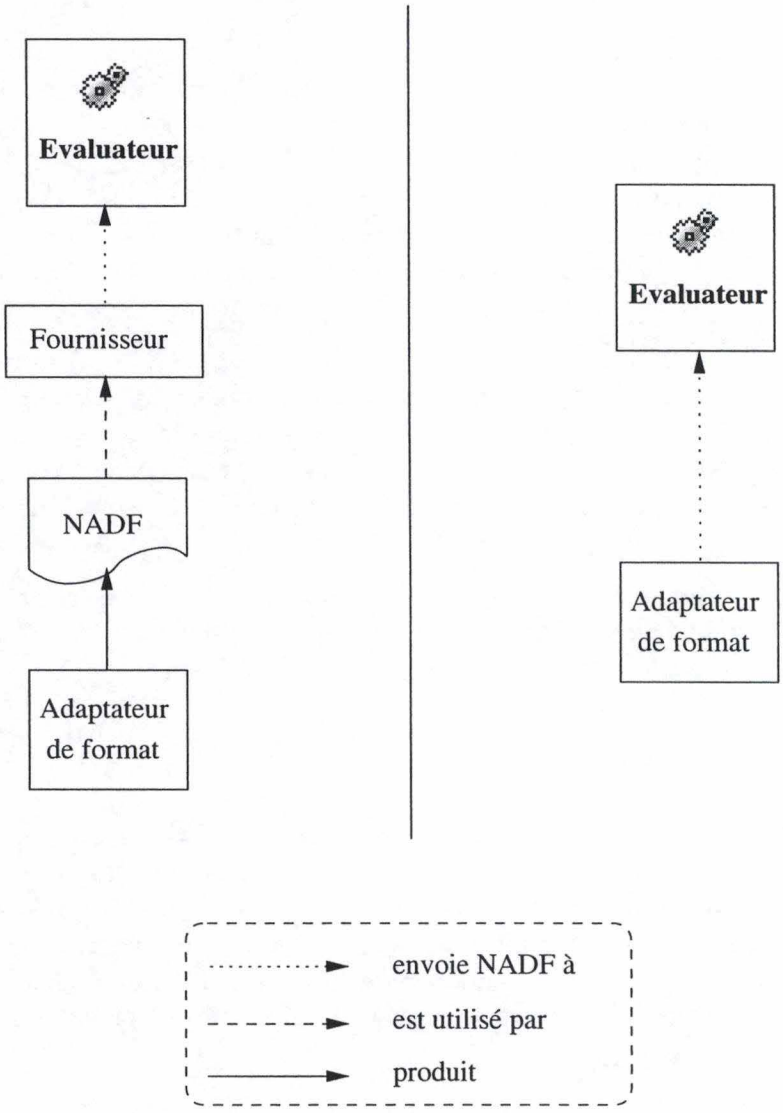


FIG. 5.1 – Les 2 modes d'utilisation de l'adaptateur de format

Malheureusement, cette connaissance n'est pas sauvegardée lors de la terminaison du programme de la console. Ainsi, si l'on quitte la console lorsqu'une analyse distribuée est en cours, celle-ci se poursuivra. Si la console est relancée par après, elle n'a plus aucune connaissance de cette analyse. Il est donc impossible d'en reprendre le contrôle. Cela implique qu'il faut purement et simplement stopper le processus d'analyse avant de quitter la console si l'on désire garder le système dans un état stable.

Nous proposons donc que la console, avant de se terminer, sauvegarde désormais la connaissance de la structure du réseau d'évaluateurs si une analyse distribuée est en cours.

Obligation d'utiliser le fournisseur

Nous avons vu que, dans le cadre d'une analyse à la volée d'une source d'audit, il était parfois souhaitable de sortir du trio classique "adaptateur de format - fournisseur - évaluateur". En particulier, on aimerait pouvoir se passer du fournisseur et établir une connexion directe, sans fichier intermédiaire, entre l'adaptateur de format et l'évaluateur. En effet, il n'est parfois pas utile de sauvegarder les données NADF dans un fichier et la sélection des enregistrements sur base de leur date de création n'a pas de sens dans le cas d'une analyse en temps réel.

Dans l'architecture actuelle, il est impossible de se passer du fournisseur. Pour remédier à cela, il faut d'abord modifier l'adaptateur de format afin qu'il puisse, au choix, enregistrer les enregistrements NADF dans un fichier ou les envoyer directement à son évaluateur. Ce point a déjà été abordé plus haut. Ensuite, il faut modifier la console ainsi que la syntaxe du fichier de description de l'environnement afin de prendre en compte ce changement.

Nous proposons donc que la console permette à l'administrateur de choisir, pour chaque source d'audit, si sa traduction NADF doit être enregistrée dans un fichier et traitée par le fournisseur ou si au contraire directement envoyée à l'évaluateur.

Hiérarchie limitée à un seul niveau

ASAX distribué permet une relation "maître-esclaves" à un niveau. Comme l'illustre la figure 5.2, cela signifie qu'un esclave ne peut pas être maître d'un esclave de niveau inférieur. Cette limitation à un niveau unique nous a pour l'instant donné pleine satisfaction mais nous pensons qu'elle pourrait devenir problématique si l'on considère le cas d'un réseau informatique composé de plusieurs centaines de machines. Il serait judicieux dès lors de pouvoir déployer les évaluateurs selon une relation maître-esclave à plusieurs niveaux comme l'illustre la figure 5.3.

L'implémentation actuelle de l'évaluateur permet déjà de passer facilement à une hiérarchie à plusieurs niveaux. Il faudra par contre revoir la conception de la console. C'est elle qui lance les exécutables et qui possède la connaissance de la composition du réseau d'évaluateurs. Il faut donc modifier la structure de représentation interne de l'analyse distribuée afin de permettre une hiérarchie "maître-esclaves" à plusieurs niveaux.

Nombre de connexions réseaux ouvertes importante

Lorsque la console a lancé à distance les différents exécutables, elle reste connectée en permanence à chaque évaluateur et fournisseur via deux connexions réseaux (*sockets*). Ces deux sockets permettent à la console d'envoyer à tout instant des ordres à un composant et permet à ce dernier d'envoyer des messages d'informations ou d'erreur à la console. Cette

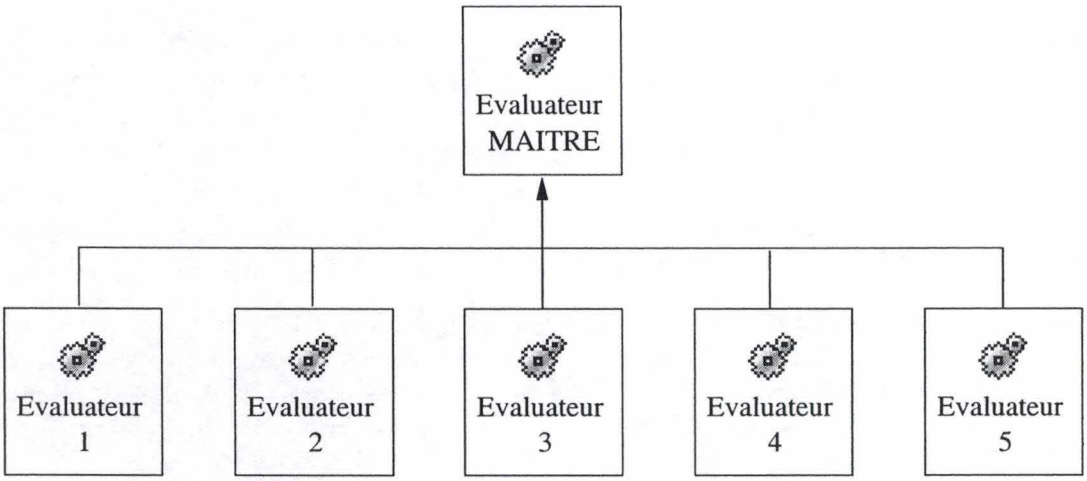


FIG. 5.2 – Relation maître-esclave à 1 niveau

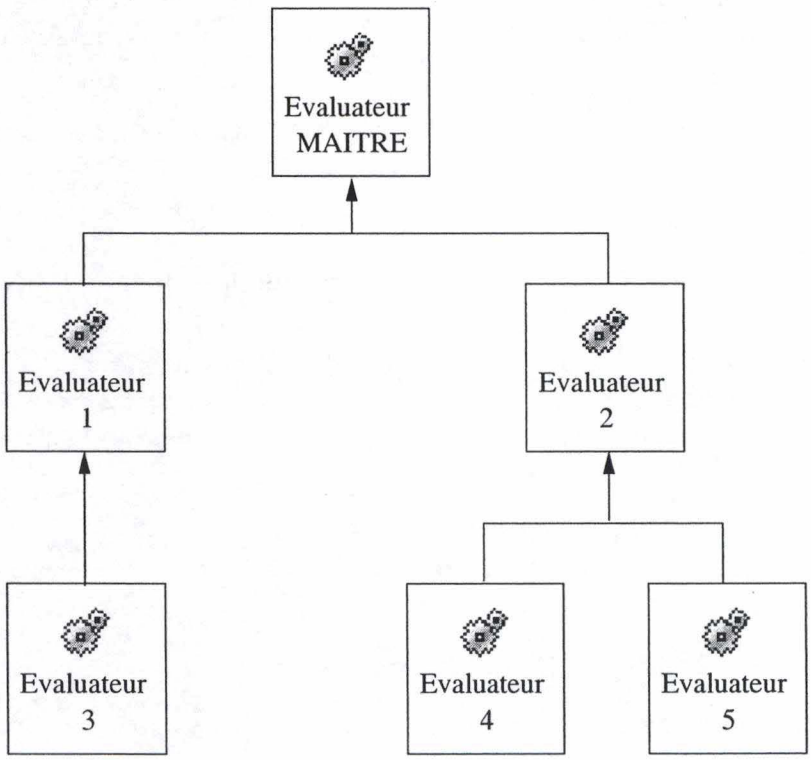


FIG. 5.3 – Relation maître-esclave à plusieurs niveaux

façon de procéder engendre un grand nombre de connexions ouvertes. Ainsi, si l'on considère un réseau de 25 machines, chaque machine hébergeant un triplet (adaptateur de format ; fournisseur ; évaluateur), la console devra maintenir 100 connexions ouvertes.

Lors du déploiement effectué durant notre stage, ce nombre élevé de connexions réseaux nous a posé problème. En effet, chaque socket est associé à un "descripteur de fichier". Par défaut, le système d'exploitation Solaris 2.7 limite à 64 le nombre de descripteurs de fichiers pouvant être ouvert par un programme. Or, le réseau étant composé de 25 machines, le nombre de descripteurs de fichiers ouverts par la console dépassait la centaine. Nous avons donc dû reconfigurer le système d'exploitation pour qu'il autorise un plus grand nombre de descripteurs de fichiers ouverts par programme.

La limite maximale étant de 1024 descripteurs par processus, cela signifie que, dans l'état actuel de l'implémentation, ASAX distribué ne peut dès lors comporter au maximum que 256 esclaves. Afin de pallier à ce problème, nous proposons de modifier légèrement l'architecture en lui ajoutant un nouveau composant : le *démon*.

Le *démon*[2] est un programme proposant des services et s'exécutant continuellement en arrière-plan. Il écoute les demandes de services sur un port. Par exemple, le démon `ftpd` écoute sur le port 21 et propose des services de transferts de fichiers.

Nous proposons donc l'utilisation d'un "démon ASAX". Ce programme sera lancé au démarrage de la machine et écoutera des demandes de connexions réseaux sur un numéro de port bien précis. Lorsque une analyse distribuée devra être exécutée, la console ne lancera plus elle-même les différents composants. Elle va établir une connexion avec le démon résidant sur une machine, lui donner les informations nécessaires pour que celui-ci puisse exécuter les composants. De plus, toutes les communications entre la console et un composant tel qu'un évaluateur passeront par le démon. De cette manière, le nombre de connexions ouvertes serait considérablement réduit.

Le fonctionnement précis du démon ainsi que son implication sur l'architecture du réseau d'analyse distribuée sont exposés à la section 5.3.

Proposition d'une nouvelle grammaire du fichier de description d'environnement

Les fonctionnalités telles que le choix du mode de sortie d'un adaptateur ou le déploiement d'évaluateurs selon arbre à plusieurs niveaux ne peuvent être exprimées dans la version actuelle du fichier descripteur d'environnement. Nous en proposons dès lors une version évoluée.

Illustrons directement par l'exemple grâce à la figure 5.2.3. Cette figure présente un exemple type de la nouvelle version du fichier descripteur d'environnement. Pour chaque évaluateur, on décrit :

- **host** : Le nom de l'hôte qui doit l'héberger
- **rules** : Le nom du fichier contenant les règles RUSSEL à appliquer
- (optionnel) Les informations concernant la source NADF (n'inclut pas les enregistrements provenant des esclaves)
  - **native** : Le format natif que doit traiter l'adaptateur de format qui lui correspond

[2]Daemon dans la terminologie anglo-saxonne

```
begin_ids
      host = backus;
      rules = general_detection;

      begin_slave
            host = aigrette;
            rules = trojan_detection;

            begin_slave
                  host = aigrette;
                  rules = evil_connection;
                  native = tcplogger;
                  mode = direct;
            end_slave

            begin_slave
                  host = pascal;
                  rules = trojan_exec;
                  native = bsm;
                  mode = file;
                  filter = [17/07/2001-*];
            end_slave
      end_slave

      begin_slave
            host = st20;
            rules = sendmail;
            native = bsm;
            mode = direct;
      end_slave
end_ids
```

FIG. 5.4 – Exemple d'une version évoluée du ficher descripteur d'environnement

• **mode :** Le mode de sortie de l'adaptateur correspondant. Si *direct*, alors l'adaptateur sera lancé et se connectera à l'évaluateur. Si *file*, c'est un fournisseur qui acheminera les enregistrements NADF à l'évaluateur.

• **filter :** (optionnel) Intervalle de temps appliqué par le fournisseur pour filtrer les enregistrements NADF (* = pas de borne)

– (optionnel) L'ensemble de ses évaluateurs esclaves

Grâce à cette nouvelle version du format de description, il est devenu possible de définir un arbre de relation "maître-esclave" à plusieurs niveaux en imbriquant les déclarations d'évaluateurs comme le montre l'exemple de la figure 5.2.3.

La grammaire BNF de ce nouveau format est représentée à la figure 5.5.

5.3 Le démon : description

Cette section a pour but de présenter le rôle du démon ASAX que nous avons proposé à la section précédente et l'incidence qu'il va avoir sur l'architecture actuelle, c'est à dire

```
<IDS_description>  ::= begin_ids <eval_body> end_ids
<eval_body>        ::= <hostname> <rules> {<source_def>}^0/1 {<slave_def>}*
<hostname>         ::= host = "any string of character" ;
<rules>            ::= rules = "any string of character" ;
<source_def>       ::= <native_format> <evaluation_mode> {<filter_def>}^0/1
<native_format>    ::= native = "any string of character" ;
<evaluation_mode>  ::= mode = <mode> ;
<mode>             ::= direct | file
<filter_def>       ::= [<date>-<date>] ;
<date>             ::= {<digit>}^2/{<digit>}^2/{<digit>}^4
                       |*
<slave_def>        ::= begin_eval <eval_body> end_eval
<digit>            ::= [0 - 9]
```

FIG. 5.5 – Proposition d'une nouvelle grammaire BNF pour le fichier descripteur d'environnement (.edf)

comment les messages vont circuler vu l'absence de connexions réseaux entre la console et les divers composants.

5.3.1 Rôles

Avant de décrire le fonctionnement du démon, il convient de définir les différents objectifs qu'il se voit attribuer dans l'IDS.

1. Tout démon ASAX doit être lancé au démarrage de la machine qui l'héberge. Ce point doit être respecté afin de pouvoir disposer constamment de machines "prêtes à l'analyse" sans se soucier des démons s'exécutant ou pas au lancement d'une analyse distribuée.
2. Le démon est chargé de lancer ou d'arrêter les différents composants sur son hôte à la demande de la console.
3. Il est chargé de maintenir une communication avec les différents composants qu'il a lancé de manière à leur communiquer des ordres et à en recevoir des messages.
4. Il joue le rôle de "borne relais" dans la circulation des messages et enregistrements NADF (voir section 5.3.2).

5.3.2 Incidence sur l'architecture réseau

Nécessité d'un identifiant par composant

Etant donné l'absence de connexions réseaux entre la console et les différents composants, il est nécessaire de concevoir un nouveau système de communication entre les différents participants à l'IDS. Le problème majeur est le suivant : comment la console peut-elle donner un ordre à un composant précis et comment un composant précis peut-il envoyer une alerte ou un message d'erreur à la console ? La réponse à cette question est l'indispensabilité de pouvoir identifier un composant parmi les autres. Un composant a donc besoin d'un identifiant et se le voit attribuer par "son" démon lors de son lancement.

Avant d'aller plus en avant dans la description de cette nouvelle architecture, résumons les idées présentées jusqu'à présent grâce à la figure 5.6. Cette dernière présente une configuration plausible d'un hôte faisant partie de l'IDS. Il dispose de 2 évaluateurs : 1 effectuant une analyse *on-line* et l'autre une analyse à partir de fichiers. Un fournisseur est donc mis en oeuvre pour ce dernier. A chacun de ces deux évaluateurs correspond bien sûr un adaptateur de format. Cependant, l'hôte dispose d'un troisième adaptateur enregistrant le résultat de sa conversion dans des fichiers. Aucune analyse n'est faite sur ces fichiers mais ceux-ci peuvent être sollicités pour une analyse ultérieure. Ces composants ont été lancés par le démon à la demande de la console. Lors de chaque lancement, le démon attribue un identifiant au composant qu'il lance et le retient de manière à pouvoir donner des ordres ultérieurement à un composant précis. Par exemple, l'administrateur peut décider de changer les règles RUSSEL à appliquer par l'évaluateur n° 2. Grâce à ce numéro, l'identifiant en fait, le démon sait à qui donner cet ordre. Il est évident, bien sûr, que le démon informe la console à quel composant correspond un identifiant de manière à ce que l'administrateur puisse reconnaître les différents évaluateurs, fournisseurs et adaptateurs qu'il a déployés.

L'identifiant ne suffit pas pour désigner un composant précis dans le réseau. En effet, les identifiants sont uniques au sein d'un hôte mais pas au sein de l'ensemble de tous les hôtes de l'IDS. C'est pourquoi, il est nécessaire d'identifier un composant en annexant le nom de son hôte à son identifiant initial. De cette manière, chaque composant est parfaitement distinct.

Plan des connexions réseaux : déploiement

Nous avons décidé que la console ne serait plus connectée avec les différents composants. Seules les connexions de type "console $\leftrightarrow$ démon" et "démon $\leftrightarrow$ démon" seraient autorisées. Les connexions entre l'évaluateur maître et ses esclaves sont permanentes dans la version actuelle d'ASAX distribué. Il convient de garder cette même stratégie dans notre nouvelle architecture pour garder un débit maximal au niveau du transfert de messages et d'enregistrement NADF entre esclaves et maîtres. Une question se pose aussitôt : comment calquer cette stratégie sur notre nouveau modèle de connexion ? La réponse est divisée en deux parties :

- La console est connectée en permanence au démon situé sur la machine hébergeant l'évaluateur maître de premier niveau.
- Le démon d'une machine hébergeant un évaluateur esclave est connecté avec le démon hébergeant le maître de cet évaluateur.

Le plus facile est d'expliquer par l'exemple. Imaginons une analyse distribuée composée de 5 évaluateurs. L'évaluateur maître se trouve sur l'hôte *backus*. Cet évaluateur dispose de 3 esclaves. Le premier se trouve sur l'hôte *st01*, les 2 restants sur *st02*. Le deuxième évaluateur hébergé par*st02* dispose lui-même d'un esclave sur l'hôte *st04*. Nous avons donc un arbre de relation "maître-esclave" à deux niveaux. Nous ne tenons pas compte des fournisseurs et adaptateurs de manière à alléger l'explication. La construction du plan des connexions réseaux est détaillée par les figures 5.7, 5.8, 5.9 et 5.10 et est composée de 4 étapes :

- **Etape 1 :** (figure 5.7) Comme le maître de premier niveau, appelons-le "maître ultime", doit être hébergé par *backus*, la console ouvre une connexion réseau avec le démon de *backus* et lui ordonne de lancer l'évaluateur. La connexion est maintenue pour une raison que nous verrons plus tard. Lors du lancement de l'évaluateur, le démon attribue à celui-ci un identifiant qu'il communique à la console.

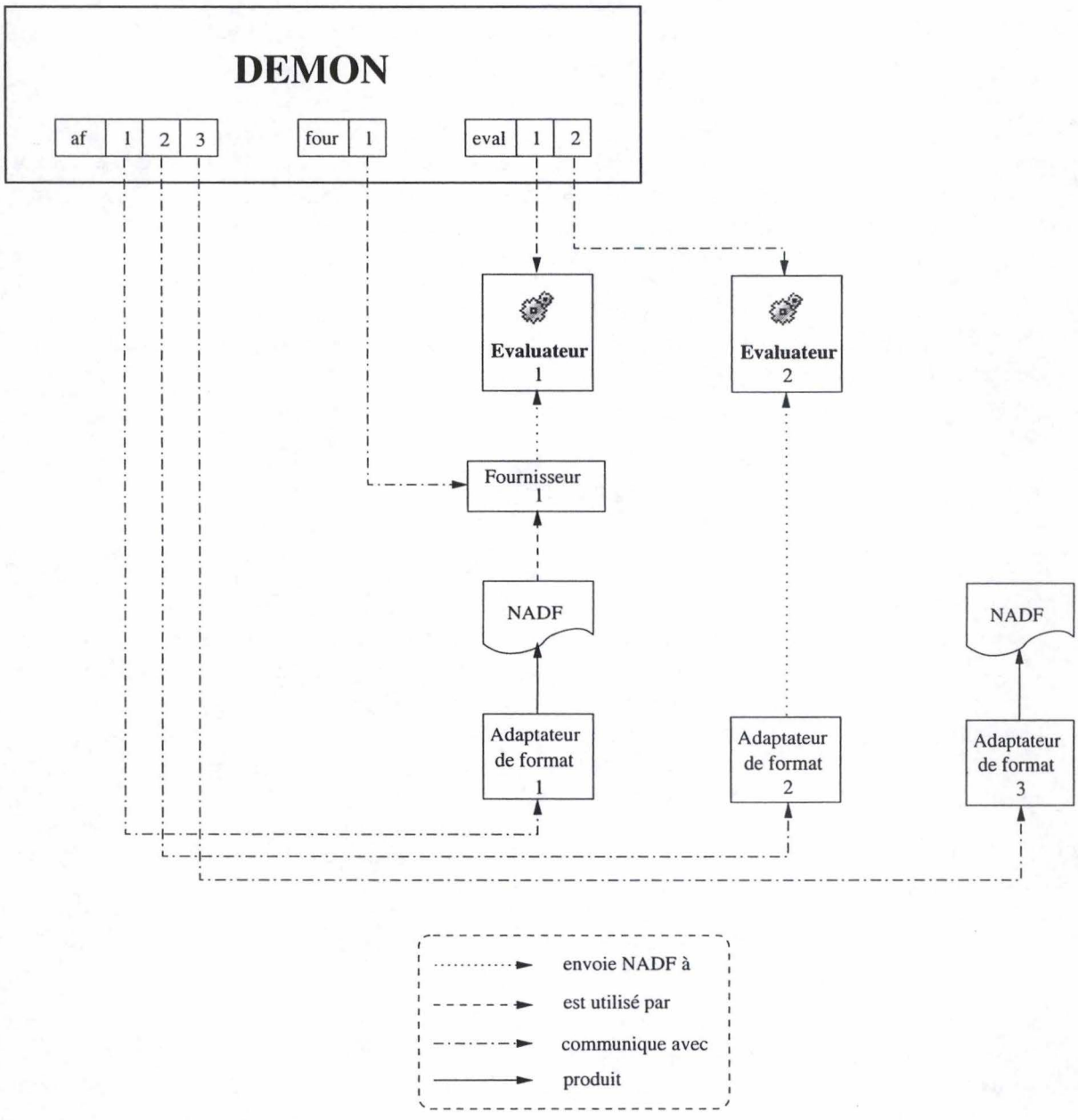


FIG. 5.6 – Architecture d'un hôte relative au démon ASAX

- **Etape 2 :** (figure 5.8) La console se connecte au démon de l'hôte *st01* et lui ordonne de lancer un évaluateur. Elle communique également au démon sur quel hôte se trouve le maître de l'évaluateur lancé ainsi que son identifiant (communiqué à la console à l'étape précédente). Le démon de *st01* ouvre alors une connexion réseau avec le démon de *backus*. Cette connexion reste permanente, elle va permettre le transfert de messages et d'enregistrement NADF. Pour chaque évaluateur qu'il a lancé, le démon garde en mémoire une référence vers le maître de cet évaluateur. Cette référence est composée du nom de la machine sur laquelle s'exécute le maître et de l'identifiant du maître permettant de le distinguer parmi les différents évaluateurs présents sur l'hôte l'hébergeant. Dans notre cas, le démon mémorise la référence au maître de son évaluateur comme étant le couple (*backus*, 1). Une fois que le démon a accompli sa mission, la console ferme la connexion qu'elle entretenait avec lui.
- **Etape 3 :** (figure 5.9) Deux évaluateurs sont lancés sur l'hôte *st02* de la même manière qu'expliqué à l'étape précédente. Remarquons que ces deux évaluateurs ont un identifiant différent.
- **Etape 4 :** (figure 5.10) La console ordonne au démon de l'hôte *st10* de lancer un évaluateur. Cet évaluateur est l'esclave du deuxième évaluateur tournant sur *st02*. Une connexion réseau est dès lors initialisée entre les démons de *st10* et *st02*. Cette connexion servira également pour le transfert de données. Afin de garantir le bon acheminement des messages ultérieurs entre l'évaluateur esclave et son maître, la référence à ce dernier est mémorisée comme étant le couple (*st02*, 2). Comme aux étapes 2 et 3, la connexion entre le démon et la console est fermée.

Dans le cas où esclave et maître résident sur le même hôte, la procédure de déploiement que nous proposons permet de gérer ce cas de manière transparente. Le démon se connecte sur lui-même et ne change pas son comportememnt en fonction de ce cas extrême. Nous n'excluons pas, néanmoins, une optimisation à ce niveau lors de l'implémentation.

Un démon n'ouvre pas plusieurs connexions réseaux avec un autre démon. Il vérifie d'abord s'il n'en a pas déjà ouvert une et dans ce cas utilise cette dernière.

L'insertion du nouveau composant qu'est le démon permet toujours la communication entre les évaluateurs. Bien que ce soient les démons qui soient connectés entre eux, chacun d'entre eux est capable de localiser le maître de chacun de "ses" évaluateurs et de permettre une communication entre ceux-ci en jouant le rôle d'intermédiaire.

Les fournisseurs et adaptateurs sont également lancés par un démon sur ordre de la console. De plus, ils se voient attribuer un identifiant de manière à pouvoir les distinguer depuis la console dans le cas d'une opération d'administration ultérieure.

Nous allons maintenant illustrer trois cas de transferts de données via les démons : l'acheminement d'un enregistrement NADF à un évaluateur maître, l'acheminement d'un message à la console et l'acheminement d'un ordre de la console à un composant précis.

Acheminement d'un enregistrement NADF à un évaluateur maître

Maintenant que nous avons établi et expliqué le mécanisme d'établissement des connexions réseaux entre les démons, nous allons illustrer ce mécanisme par une utilisation concrète des communications entre évaluateurs, c'est-à-dire, le transfert d'un enregistrement NADF par un évaluateur à son maître.

Lorsque qu'un évaluateur désire envoyer un enregistrement NADF à son maître, il l'exprime explicitement à "son" démon. Grâce aux informations qu'il a mémorisées lors du

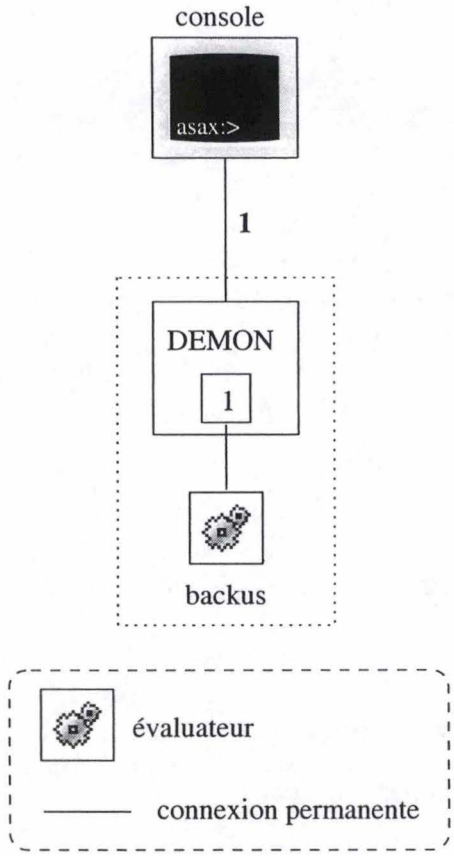


FIG. 5.7 – Exemple de déploiement d'une analyse distribuée via le démon : étape 1

déploiement, celui-ci sait sur quelle machine se trouve le maître en question et connaît l'identifiant permettant de le distinguer parmi les autres évaluateurs résidant sur cette machine. Ainsi, la figure 5.11 illustre l'exemple d'un évaluateur présent sur *backus* envoyant un enregistrement NADF à son maître présent sur l'hôte *aigrette*. L'évaluateur esclave donne à "son" démon un enregistrement NADF et lui fait savoir qu'il désire l'envoyer à son maître. Le démon sait que le maître recherché réside sur l'hôte *aigrette* et est identifié comme étant le n° 2. Il communique dès lors au démon de *aigrette* l'enregistrement NADF et désigne l'évaluateur destination comme étant le n° 2. Le démon "cible", c'est-à-dire celui résidant sur *aigrette* prend en charge l'enregistrement NADF et le transmet à "son" évaluateur n° 2.

Note : Comme nous l'avons dit précédemment, il n'est pas nécessaire de gérer spécialement le cas où le maître et l'esclave résident sur le même hôte. Le démon envoie l'enregistrement à lui-même et le transmet au maître.

Acheminement d'un message à la console

Quand un composant désire envoyer un message à la console (`println()` RUSSEL ou message d'erreur par exemple), il lui est impossible de le communiquer directement suite à notre modification de l'architecture. Le message doit donc être acheminé via les démons jusque la console. Cet acheminement n'est pas aussi simple qu'il y paraît. Imaginons un démon connecté à trois autres démons. Un message en destination de la console lui parvient à partir du premier démon. Sur quelle base peut-il décider à quel démon parmi les deux

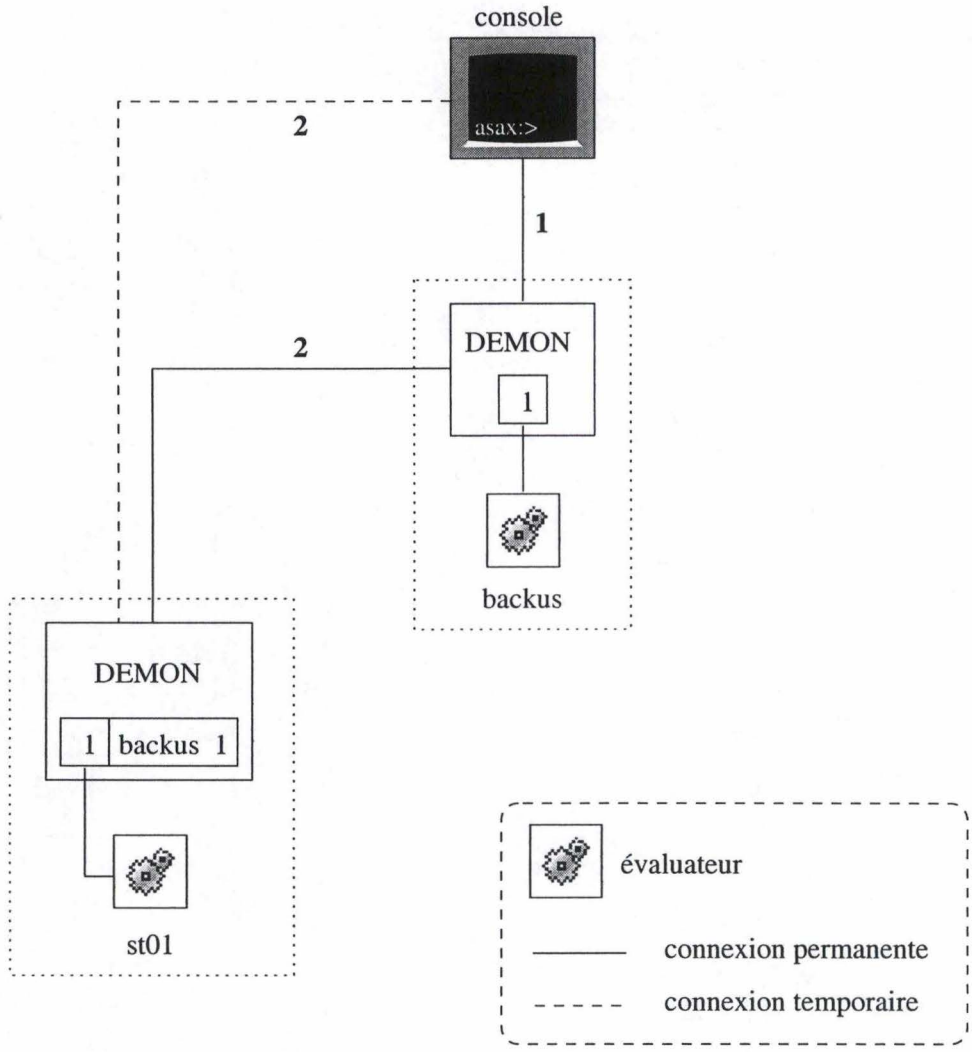


FIG. 5.8 – Exemple de déploiement d'une analyse distribuée via le démon : étape 2

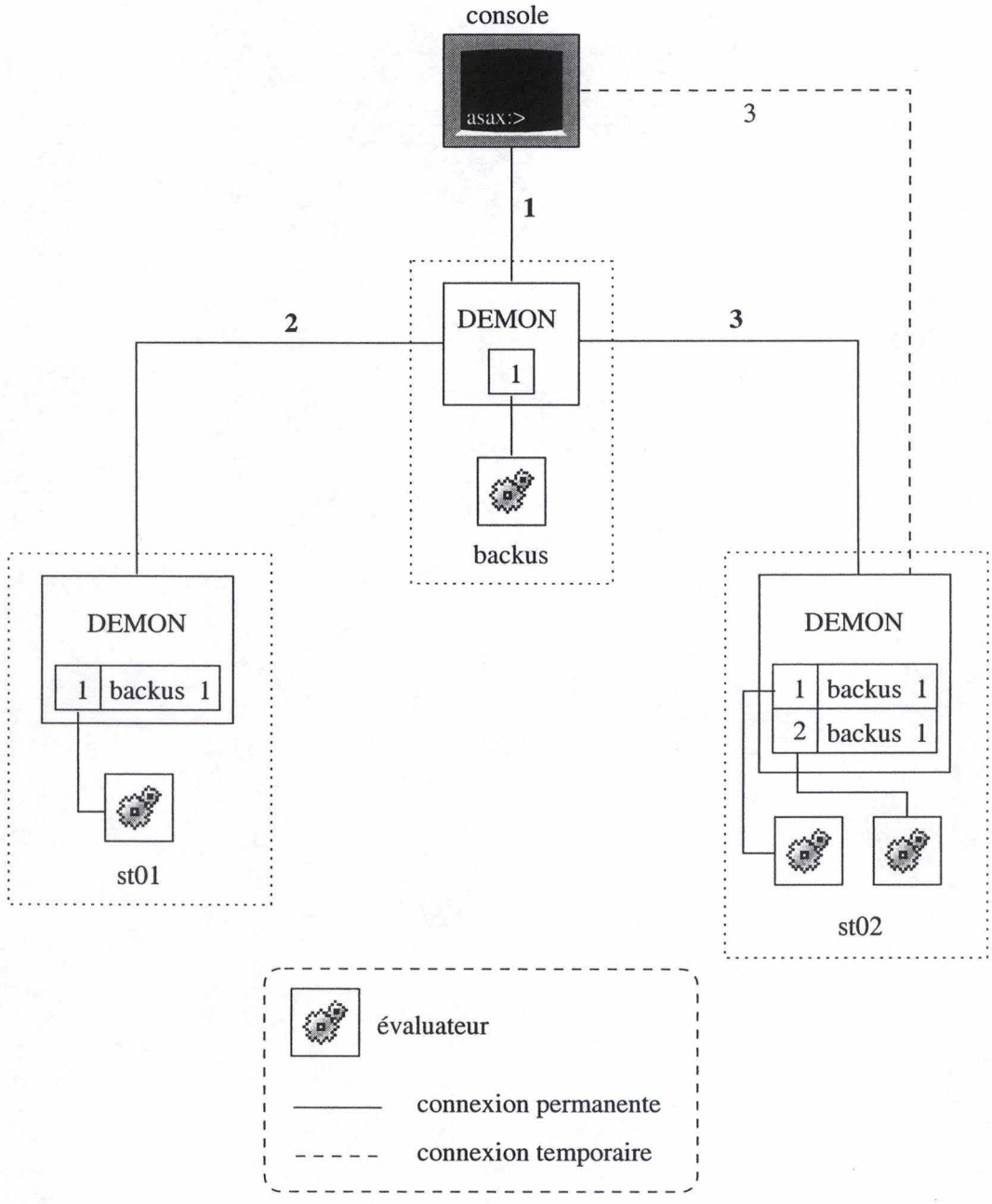


FIG. 5.9 – Exemple de déploiement d'une analyse distribuée via le démon : étape 3

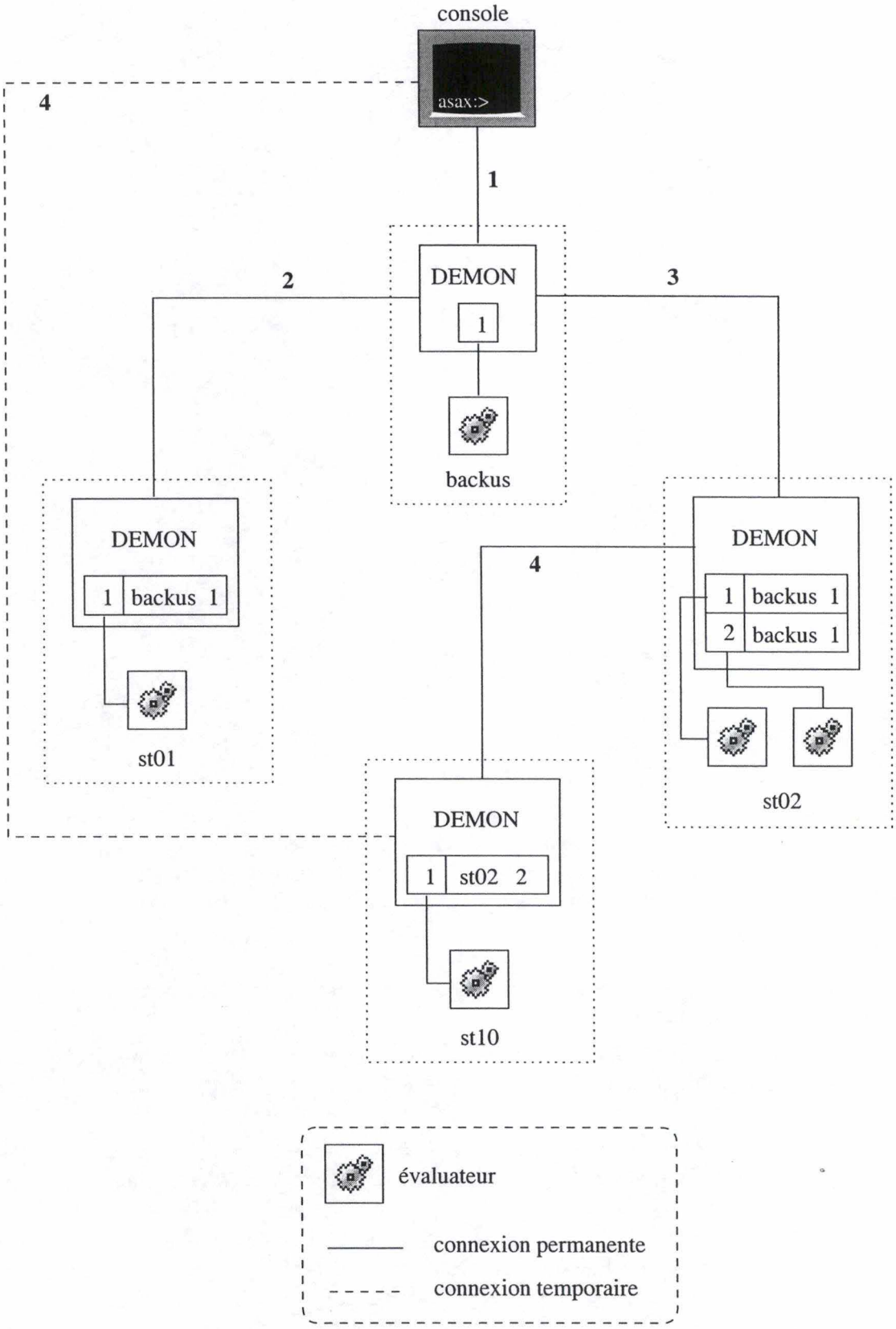


FIG. 5.10 – Exemple de déploiement d'une analyse distribuée via le démon : étape 4

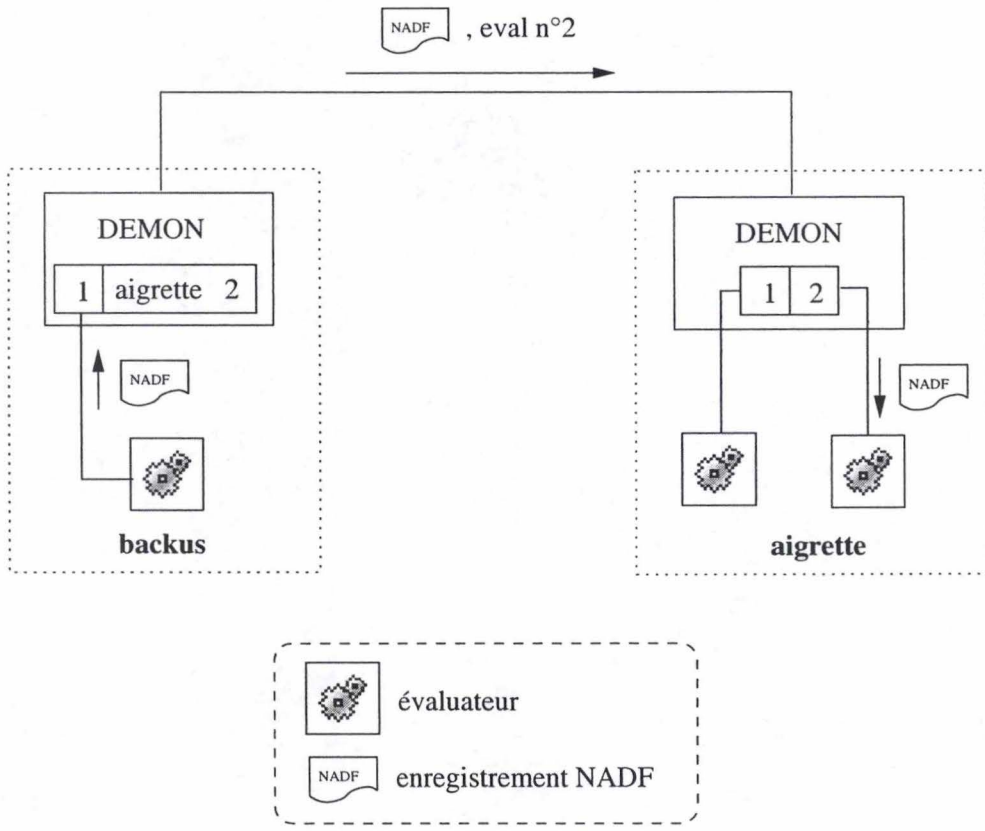


FIG. 5.11 – Acheminement d'un enregistrement NADF à un évaluateur maître

autres il doit passer le message ? Notre proposition est simple, le message doit remonter l'arbre des relations "maître-esclave" jusqu'au maître ultime qui le transmet à la console.

Ainsi, prenons l'exemple illustré à la figure 5.12 où l'évaluateur n° 1 de l'hôte *st10* désire envoyer un message à la console. Afin d'atteindre cette dernière, le message doit remonter l'arbre de relation "maître-esclave". De cette manière, il est assuré d'atteindre le démon correspondant au maître ultime et qui est constamment connecté à la console. Décrivons l'acheminement étape par étape :

1. L'évaluateur n° 1 de l'hôte *st10* remet son message au démon en lui explicitant qu'il est destiné à la console.
2. Le démon connaît la référence du maître de l'évaluateur émetteur. Dans notre exemple, il s'agit du couple (*st02*, 2). Le démon transmet dès lors le message au démon résidant sur *st02*.
3. Le démon de *st02* reçoit le message. Une question se pose dès lors : comment peut-il savoir à quel démon, entre celui de *backus* et *st20*, il doit envoyer le message ? Tout simplement par ce que ce dernier doit suivre les relations "maître-esclave" jusqu'au sommet de l'arbre, c'est-à-dire le maître ultime. Le démon de l'hôte *st10* communique alors l'identifiant de l'évaluateur correspondant au noeud de l'arbre où le message est arrivé (ici, l'évaluateur n° 2). Le démon de l'hôte *st02*, une fois en possession de cette information, consulte sa table de référence et regarde qui est le maître de l'évaluateur n° 2. Il s'agit de l'évaluateur n° 1 sur *backus*. Le message est donc envoyé au démon de *backus*.
4. Etant donné que le démon de *backus* est connecté à la console, le message peut donc être livré à destination. Le fait de maintenir la connexion ouverte entre le démon

hébergeant le maître ultime et la console est maintenant clairement justifié.

Acheminement d'un ordre de la console à un composant précis

A chaque composant lancé, le démon lui attribue un identifiant qu'il communique à la console. Grâce à cette "collection" d'identifiants, la console peut donc distinguer chaque composant d'un autre parmi l'ensemble contribuant à l'IDS. Prenons l'exemple d'un évaluateur identifié en tant que n° 3 sur l'hôte *backus*. L'administrateur désire communiquer un ordre de changement de configuration à cet évaluateur. Comme la console connaît son identifiant et l'hôte sur lequel il réside, elle ouvre dès lors une connexion avec le démon de *backus* et lui transfère un message de la forme : "Donne tel ordre à l'évaluateur n° 3". Une fois cet ordre transmis, la connexion est fermée. Cet exemple est bien évidemment généralisable aux adaptateurs de format et fournisseurs.

5.3.3 Analyses distribuées en parallèle

Jusqu'à présent nous n'avons considéré que des situations où une seule analyse distribuée était en cours d'exécution. Toutefois, ASAX est capable d'en effectuer plusieurs en parallèle. Dans ce cas, la console est connectée à tous les démons hébergeant les maîtres ultimes de chacune des analyses distribuées. Les composants étant identifiés par les démons, il n'y a donc pas lieu de les confondre. En effet, un identifiant ne se répète pas d'une analyse à l'autre sur le même hôte. Un hôte, ou plus précisément, un démon peut donc contribuer à plusieurs analyses en même temps. La seule différence à noter sera le chemin parcouru par les messages destinés à la console selon l'appartenance d'un composant à une analyse ou à une autre. Ce chemin dépend en effet de l'arbre de relation "maître-esclave" propre à l'analyse.

5.4 Fonctionnalités offertes par le poste d'administration

Suite aux changements apportés dans l'architecture via l'introduction du démon, les commandes fournies par la console ne sont plus valides. En effet, cette dernière n'est plus connectée directement avec les composants mais avec des démons. Il est donc nécessaire d'étendre les commandes en leur annexant des informations supplémentaires de façon à communiquer avec le bon démon et ce dernier avec le bon composant. Par exemple, si l'on désire envoyer un ordre à un évaluateur, la version actuelle de la console envoie cet ordre via une connexion directe avec l'évaluateur cible. Par contre, dans notre version, la console n'est pas connectée à cet évaluateur. Il convient donc d'annexer à la commande le nom de l'hôte sur lequel il se trouve de manière à acheminer l'ordre jusqu'au démon correspondant et son identifiant au sein de cet hôte, afin que le démon puisse communiquer avec le bon évaluateur. Nous allons donc dans cette section présenter les versions "étendues" des commandes offertes par la console.

Nous proposons également de nouvelles commandes de manière à améliorer la flexibilité de l'administration d'ASAX distribuer à distance. Par exemple, nous proposons une commande permettant de rajouter un évaluateur dans une analyse distribuée.

Durant l'utilisation de la version distribuée d'ASAX lors de notre stage, nous avons constaté l'existence de commandes "console" spécifiques au système d'audit BSM. Nous jugeons que ces fonctionnalités ne devraient pas être reprises par la console car, si ASAX tend à vouloir tout gérer, il subira constamment des mises à jour en fonction de l'évolution des

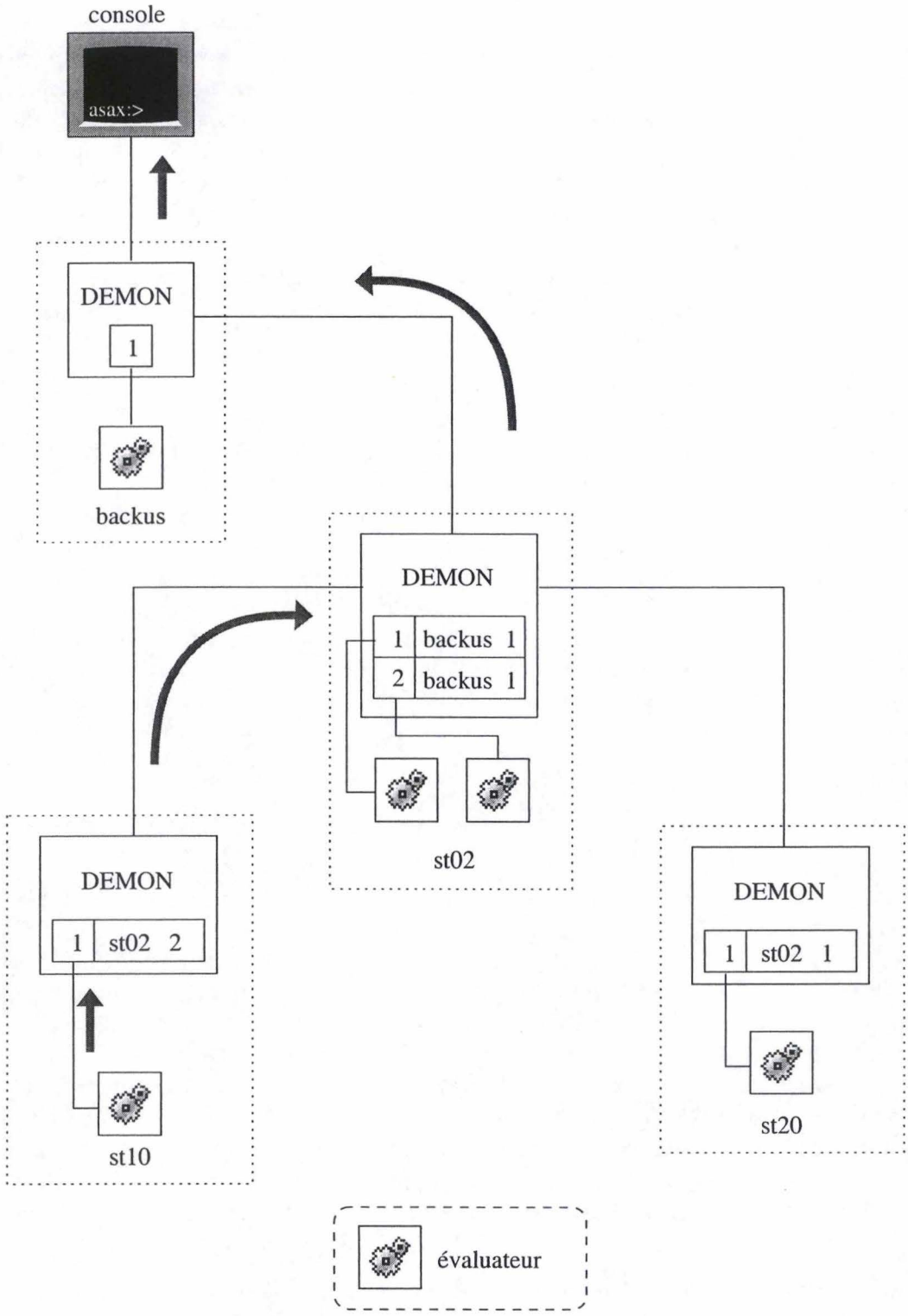


FIG. 5.12 – Acheminement d'un message à la console

différents audits qu'il désire administrer. C'est pourquoi, nous jugeons préférable qu'ASAX n'ait aucune interaction avec les différents systèmes d'audit.

Nous proposons par conséquent de déplacer ces fonctionnalités dans des scripts fournis avec ASAX de manière à le laisser aussi indépendant que possible des systèmes d'audit tout en proposant à l'utilisateur un confort d'utilisation accru.

5.4.1 Lancement d'une analyse distribuée

```
run environnement_description_file
```

Cette commande ne propose rien de plus que l'actuelle. Cependant, nous avons remarqué que la console dont nous disposions pendant notre stage ne pouvait gérer plusieurs analyses distribuées s'exécutant en parallèle (deux `run` consécutifs par exemple). Par conséquent, nous étions obligés de disposer d'une console pour chacune des analyses distribuées. C'est pourquoi nous insistons sur le fait que la commande `run` introduite dans cette section lance une analyse distribuée qui doit être gérée en parallèle et en parfaite harmonie avec les autres analyses en cours et ce, à partir de la même console.

Bien sûr, afin de permettre de cibler l'une ou l'autre analyse distribuée dans le cadre de certaines commandes, il convient de les identifier par un numéro identifiant qui serait attribuer lors du lancement de la commande `run`.

`environnement_description_file`	Fichier décrivant la configuration de l'analyse distribuée à lancer

5.4.2 Arrêt d'une analyse distribuée

```
halt ids_id
```

Cette commande arrête l'exécution d'une analyse distribuée identifiée par le paramètre `ids_id`. Tous les composants participant à cette analyse sont arrêtés, excepté les démons. Nous avons vu à la section 5.2.2, l'existence de deux types d'adaptateur de format : l'un écrivant stockant les enregistrements NADF dans des fichiers, l'autre les envoyant directement à un évaluateur. Il est évident que le premier n'appartient pas à une analyse distribuée particulière. En effet, l'administrateur peut très bien lancer un tel évaluateur, sans pour autant lancer une analyse, afin de rassembler des données NADF qui serviront à un traitement ultérieur. De plus, plusieurs analyses parallèles peuvent puiser dans les fichiers produits par cet évaluateur. Il est donc évident qu'il ne soit pas arrêté par la commande `halt`.

`ids_id`	Identifiant de l'analyse à interrompre

5.4.3 Liste des analyses distribuées

```
list_ids
```

Cette commande affiche la liste des identifiants des analyses distribuées en cours d'exécution.

5.4.4 Affichage d'informations sur le système

```
info ids_id | hostname
```

Cette commande affiche une liste d'information selon l'option choisie par l'administrateur :

1. Soit l'administrateur désire obtenir des informations sur une analyse distribuée précise.
2. Soit l'administrateur désire obtenir des informations sur un hôte précis.

Dans le cas de la première option, les informations fournies sont regroupées par hôte et consistent en ceci :

- *Le nom de l'hôte*
- *Pour chaque évaluateur appartenant à l'analyse distribuée donnée*
  - son identifiant
  - le fichier de règles RUSSEL appliquées
  - l'identifiant et l'hôte de son évaluateur maître
  - les identifiants et hôtes de ses évaluateurs esclaves
  - le format natif des données qu'il analyse
  - (optionnel) : dans le cas où l'évaluateur est directement connecté à un adaptateur de format, l'identifiant de ce dernier
  - (optionnel) : s'il dispose d'un fournisseur, l'identifiant de ce dernier

Dans le cas de la deuxième option, les informations fournies consistent en ce qui suit :

- *Pour chaque évaluateur*
  - son identifiant
  - l'identifiant de l'analyse à laquelle il participe
  - le fichier de règles RUSSEL appliquées
  - l'identifiant et l'hôte de son évaluateur maître
  - les identifiants et hôtes de ses évaluateurs esclaves
  - le format natif des données qu'il analyse
  - (optionnel) : dans le cas où l'évaluateur est directement connecté à un adaptateur de format, l'identifiant de ce dernier
  - (optionnel) : s'il dispose d'un fournisseur, l'identifiant de ce dernier
- *Pour chaque fournisseur*
  - son identifiant
  - l'identifiant de l'évaluateur qu'il fournit
  - l'intervalle de temps appliqué en tant que filtre
- *Pour chaque adaptateur de format*
  - son identifiant
  - le format natif des données qu'il traduit
  - (optionnel) : dans le cas où il est connecté à un évaluateur, l'identifiant de ce dernier
  - (optionnel) : dans le cas où il écrit sa conversion dans un ou plusieurs fichiers, la taille maximale d'un fichier NADF produit.

`ids_id`	Identifiant de l'analyse sur laquelle des informations sont désirées
`hostname`	Nom de l'hôte sur lequel des informations sont désirées

5.4.5 Choix de la taille maximale d'un fichier au format NADF

```
NADF_size hostname fa_id size
```

Cette commande permet à l'administrateur de modifier en cours d'exécution, pour un adaptateur de format donné, la taille maximale qu'un fichier NADF peut atteindre. Rappelons qu'une fois cette taille atteinte, l'adaptateur ferme le fichier et continue son stockage dans un nouveau fraîchement créé.

`hostname`	nom de l'hôte sur lequel se trouve l'adaptateur concerné
`fa_id`	identifiant de l'adaptateur concerné
`size`	nouvelle taille maximale en octets d'un fichier NADF

5.4.6 Arrêt d'un adaptateur de format

```
fa_stop hostname fa_id
```

Cette commande offre la possibilité d'arrêter l'exécution d'un adaptateur de format donné. Dans le cas où il s'agit d'un adaptateur directement connecté à un évaluateur, celui-ci reçoit également un ordre d'arrêt car cela n'aurait aucun sens qu'il continue à attendre des informations venant de l'adaptateur arrêté.

`hostname`	nom de l'hôte sur lequel se trouve l'adaptateur concerné
`fa_id`	identifiant de l'adaptateur concerné

5.4.7 Lancement d'un adaptateur de format

```
fa_run hostname native_format [size]
```

Cette commande permet de lancer des adaptateurs de format stockant le résultat de leur conversion dans des fichiers. Elle permet de rajouter un de ces composants dynamiquement lors de l'exécution de l'IDS. L'utilisateur doit fournir le format natif à traduire et, optionnellement, la taille maximale des fichiers NADF de sortie. Si l'utilisateur ne fournit pas cette dernière, une taille par défaut est utilisée.

Note : Afin de lancer l'adaptateur de format correspondant au format natif donné, il convient d'adopter la règle de nomenclature suivante au niveau des adaptateurs de format : le nom des exécutables de chacun des adaptateurs est préfixé de `fa_` et suivi par le nom du format natif qu'il traduit. Par exemple, l'adaptateur traduisant le format BSM s'appellera `fa_bsm`. De cette manière, il suffit de concaténer le format fourni dans la commande avec `fa_` pour obtenir le nom du bon adaptateur.

`hostname`	nom de l'hôte sur lequel l'adaptateur doit être lancé
`native_format`	format natif résultant des processus d'audit (ex : BSM)
`size`	taille maximale en octets d'un fichier NADF

Application d'un nouvel ensemble de règles RUSSEL

```
russel hostname eval_id russel_file
```

Dans le cas où l'administrateur modifie des règles RUSSEL ou en crée de nouvelles, il lui est possible de les passer aux évaluateurs en cours d'analyse sans pour autant les arrêter et les redémarrer.

`hostname`	nom de l'hôte sur lequel se trouve l'évaluateur concerné
`eval_id`	identifiant de l'évaluateur concerné
`file`	fichier dans lequel se trouve les nouvelles règles à appliquer

5.4.8 Ajout d'un nouvel évaluateur

```
eval_add hostname russel_file -m hostname eval_id -d native_format

eval_add hostname russel_file -m hostname eval_id -f native_format [time_filter]
```

Cette commande offre la possibilité à l'administrateur de lancer un évaluateur supplémentaire au sein de l'IDS. Pour obtenir un évaluateur opérationnel, il convient bien sûr de lui administrer un ensemble de règles RUSSEL à appliquer et de lui désigner un évaluateur maître. Il est également nécessaire de décider si l'analyse se fera *on-line* ou à partir de fichiers stockés. Dans le premier cas, l'adaptateur correspondant au format natif fourni sera lancé automatiquement. Dans le second, ce sera un fournisseur adéquat qui sera lancé automatiquement. Dans ce dernier cas, l'administrateur peut définir un intervalle de temps à appliquer par le filtrage des enregistrements NADF envoyés à l'évaluateur.

En toute logique, ce nouvel évaluateur participe à l'analyse distribuée à laquelle son maître participe.

`hostname`	nom de l'hôte sur lequel l'évaluateur va être lancé
`russel_file`	fichier dans lequel se trouve les règles RUSSEL à appliquer
`-m hostname eval_id`	le nom de l'hôte et l'identifiant de l'évaluateur maître
`-d native_format`	mode *on-line*, un adaptateur du format `native_format` est lancé et connecté à l'évaluateur
`-f native_format`	un fournisseur est lancé afin d'envoyer à l'évaluateur les enregistrements NADF correspondant au format natif fourni
`time_filter`	intervalle de temps à appliquer par le filtrage des enregistrements NADF

5.4.9 Arrêt d'un évaluateur

```
eval_stop hostname eval_id
```

Cette commande permet à l'administrateur d'arrêter l'exécution d'un évaluateur choisi. Dans le cas où un évaluateur maître est destiné à être supprimé, il convient de décider de l'avenir de ses esclaves. Par conséquent nous avons décidé que tous ses esclaves directs et indirects s'arrêteraient également. Bien sûr, pour chaque évaluateur, son fournisseur ou adaptateur "conjoint" est aussi interrompu.

`hostname`	nom de l'hôte sur lequel se trouve l'évaluateur à interrompre
`eval_id`	identifiant de l'évaluateur à arrêter

5.5 Conclusion

Malgré les lacunes d'implémentation, la version distribuée d'ASAX est fonctionnelle. Cependant, l'administrateur attache une grande importance à la transparence et la flexibilité acompagnant les fonctionnalités offertes. Par exemple, il peut désirer étendre le langage RUSSEL avec ses propres routines afin d'optimiser la détection de comportements suspects. Ou encore, il peut désirer modeler la configuration de son IDS selon les ressources dont il dispose. Ce chapitre a présenté quelques propositions d'adaptations et d'évolutions de la version distribuée d'ASAX. Par exemple, l'utilisation de librairies dynamiques pour étendre RUSSEL est un exemple typique de gain de transparence. La version actuelle d'ASAX distribué impose une certaine forme dans la configuration de l'IDS (stockage dans des fichiers NADF, source d'audit systématiquement BSM). Nous avons vu que les composants sont indépendants de ce problème. Ce dernier est centré dans le poste de commande qu'est la console. Nous avons donc proposé une évolution de cette dernière tout en lui administrant de nouvelles fonctionnalités afin de permettre à l'administrateur de modifier dynamiquement la configuration de l'IDS de manière souple.

Cette augmentation de flexibilité couplée avec l'introduction du démon, ce dernier permettant une certaine "discipline" au niveau des communications inter-composants, est le début de la procédure à suivre afin de mener ASAX vers le statut d'application "clef en main" plutôt que "prototype". A partir de ce moment, ASAX pourra être utilisé en tant qu'outil de sécurité à grande échelle et l'établissement d'une base de règles de détection deviendra alors l'intérêt majeur.

Chapitre 6

Sécurisation d'ASAX

6.1 Introduction

Le chapitre 1 a montré que de nombreuses attaques exploitent des vulnérabilités présentes dans certains programmes. Ces failles sont pour la plupart la conséquence d'erreurs de programmation ou de négligences de la part des programmeurs. Lors du développement d'un programme, il faut en effet s'assurer de sa robustesse : l'utilisateur ne doit pas pouvoir le "planter" ; les entrées doivent systématiquement être vérifiées afin d'empêcher tout risque de *buffer overflow*, ...

De plus, les programmes comportant plusieurs composants communiquant entre eux doivent s'assurer de l'authenticité des communications inter-composants. Un intrus peut en effet se faire passer pour l'un de ces modules et compromettre le système.

ASAX ne doit pas déroger à ces règles de développement. Il serait en effet fâcheux qu'un système de détection d'intrusion puisse être détourné de son utilisation normale et serve à attaquer une machine. Comme précisé dans [Mou97], la sécurité n'a pas été prise en compte lors du développement du prototype d'ASAX distribué. De même, dans les quelques améliorations proposées au chapitre 5, nous n'avons pas abordé la question de la sécurité. Ce chapitre tente donc de combler ces lacunes.

Nous présentons dans la première section de ce chapitre un tour d'horizon des attaques susceptibles d'être effectuées contre ASAX. Ensuite, nous mettrons en évidence quelques points auxquels il faudra porter attention lors d'une mise à jour future du code d'ASAX et de son utilisation quotidienne.

6.2 Panorama des attaques possibles sur ASAX

En nous mettant dans la peau d'une personne voulant attaquer un réseau protégé par ASAX, nous tenterons de voir s'il est possible de contourner ou de neutraliser l'IDS. Nous essayerons également de tirer profit de la présence de l'IDS pour récupérer de précieuses informations sur le système et ses utilisateurs. Nous tenterons aussi de prendre le contrôle de l'IDS et de le retourner contre le système.

Dans la suite de cette section, nous explorons différents angles d'attaque et, pour chacun d'eux, tentons de dresser une liste de tous les détournements possibles.

6.2.1 Interception des communications inter-composants

Les différents composants d'ASAX distribué communiquent entre eux de manière non sécurisée. Il est donc intéressant pour un attaquant d'écouter le trafic réseau et de reconstituer les "dialogues" entre composants. Il peut alors récupérer beaucoup d'informations qui peuvent lui servir pour préparer une attaque. Donnons quelques exemples :

- Il peut obtenir des informations sur la structure du réseau d'évaluateurs ASAX. S'il écoute le trafic depuis l'initialisation de l'analyse, il est capable de connaître le nom de chaque machine hébergeant un évaluateur et peut reconstituer les relations "maître-esclaves" entre ces évaluateurs. Cela lui permet de connaître les machines ne faisant pas partie de l'analyse. Il sera donc tenté de les attaquer en premier lieu.
- Il peut également obtenir des informations sur le comportement de l'IDS. Les noms des fichiers contenant les règles RUSSEL à exécuter par les évaluateurs sont en général assez explicites et donnent de précieuses informations sur les règles de détection effectivement en cours d'utilisation.
  Les alarmes et informations transmises par les évaluateurs peuvent également donner des indications sur les règles utilisées et sur les éventuels seuils de détection au-delà desquels la répétition d'une action est considérée comme une attaque (cfr. l'exemple des 3 logins infructueux.).
- Les événements systèmes encapsulés dans les enregistrements NADF transmis entre évaluateurs contiennent également de précieux renseignements. Ils peuvent nous donner des noms d'utilisateurs valides, leur numéro UID, ...
  Plus sournoisement, ces informations peuvent fournir des détails quant à l'utilisation du matériel : horaire du personnel, programmes lancés fréquemment, ... L'attaquant peut donc à distance avoir une idée de l'horaire d'occupation des locaux et préparer son attaque physique en conséquence. De même, en connaissant les programmes les plus utilisés, il peut sélectionner un Cheval de Troie qui aura plus de chance d'être lancé par sa victime.
- Dans la version distribuée d'ASAX, pour pouvoir lancer à distance les différents composant, la console a besoin d'un nom d'utilisateur et d'un mot de passe. La méthode d'exécution à distance `rexec()` utilisée transmet ces informations en clair sur le réseau. Il est donc facile de récupérer ces informations en écoutant le trafic généré à l'initialisation d'ASAX.
  L'introduction des démons résout en partie ce problème car les programmes ne sont plus lancés directement à distance par la console. Cette dernière contacte en effet les démons qui se chargeront alors d'exécuter les composants localement. Cette exécution locale ne nécessite pas de nom d'utilisateur ou de mot de passe. Cependant, cette solution amène un autre problème : comment être certain que les ordres donnés aux démons proviennent effectivement de la console ? Ce point sera abordé ci-après.

6.2.2 Usurpation des communications inter-composants

Quand les composants communiquent entre eux, ils ne vérifient pas l'identité de leurs correspondants. On peut alors imaginer qu'un attaquant profite de cette situation.

S'il connaît le protocole de communication utilisé par les éléments d'ASAX distribué, il peut se connecter sur l'un d'entre eux en se faisant passer pour un autre composant. Il peut alors corrompre le système. Explorons les possibilités d'attaques :

- L'attaquant peut se faire passer pour la console et prendre le contrôle total du système. Il sera alors en mesure d'ajouter ou de supprimer des évaluateurs, de changer les

règles utilisées ou encore arrêter tout le processus d'analyse. Grâce à ce contrôle sur le processus d'évaluation, l'attaquant peut aisément dissimuler ses traces en neutralisant l'analyse sur les machines qu'il compte attaquer.

- En se faisant passer pour un évaluateur, l'attaquant peut fortement perturber le comportement du système. Il lui suffit pour cela d'envoyer à répétition un nombre important de fausses alarmes. Il peut pendant ce temps effectuer l'attaque de son choix. Même si elle est effectivement détectée, il y a fort à parier qu'elle passera inaperçue auprès de l'administrateur système, noyée parmi la masse des fausses alertes générées.
- Un attaquant peut jouer un mauvais tour à une personne en la faisant passer pour un individu mal intentionné. Il doit pour cela usurper l'identité d'un évaluateur et forger de toute pièce des alarmes dont le contenu présente la victime innocente comme étant l'auteur de différentes attaques.

6.3 Sécurisation des composants et des communications entre ceux-ci

La liste des attaques présentées dans la section précédente est longue, mais faut-il pour autant s'en inquiéter ? En effet, ces attaques ne sont pas aisément réalisables. Elles demandent du temps, une maîtrise des protocoles de communication ainsi qu'une connaissance parfaite du fonctionnement d'ASAX.

Nous estimons cependant qu'il est indispensable d'éliminer toutes les failles de sécurité, même si elles sont difficilement exploitables. Dès lors, il n'est pas acceptable que les communications entre composants soient non-chiffrées et non-authentifiées. Se pose alors la question : quel protocole sécurisé utiliser ?

Choisir un protocole effectuant un chiffrage des données n'est pas suffisant, il faut également que ce protocole permette d'effectuer une authentification des composants, ceci afin d'empêcher une "attaque par rejeu". Illustrons ce type d'attaque par un exemple :

Un intrus enregistre le trafic réseau et, en particulier, les communications entre la console et le démon se situant sur l'hôte hébergeant l'évaluateur maître. Ces communications sont chiffrées ; l'intrus n'en comprend donc pas le sens. Imaginons qu'il remarque à un moment donné que la console envoie un message à différentes machines et, après émission de ce message, qu'il n'y a plus aucune activité de transmission entre les différents composants. Il y a donc fort à parier que ces derniers messages chiffrés étaient en réalité un ordre d'arrêt de l'analyse distribuée.

Lorsque l'analyse sera relancée, l'intrus peut se faire passer pour la console et rejouer les derniers messages chiffrés qu'il a enregistrés. Comme il n'y a pas de mécanisme d'authentification, les composants penseront que les ordres viennent effectivement de la console et l'analyse distribuée s'arrêtera à nouveau.

On le voit par cet exemple, la sécurisation des communications inter-composants passe par l'utilisation d'un protocole proposant à la fois un chiffrement des messages et une authentification des parties. Choisir un protocole de chiffrement est une tâche délicate qui nécessite une étude comparative basée sur plusieurs critères ; il y a bien sûr la puissance de chiffrement, qui est en général liée à la longueur de la clef ; il y a l'investissement en infrastructure induit par les protocoles d'authentification qui font en général appel à un serveur de clefs, appelé "tiers de confiance" ; la puissance de calcul nécessaire pour effectuer les actions de chiffrage/déchiffrage doit également être prise en compte. Enfin, n'oublions pas que de nombreux pays considèrent encore les algorithmes de chiffrement comme une

arme de guerre et en limitent donc l'importation, l'exportation ainsi que la puissance de chiffrement.

Cette comparaison des protocoles existants dépasse le cadre de ce mémoire, mais devra impérativement être effectuée avant de pouvoir utiliser ASAX à grande échelle.

Si la sécurisation des liaisons entre composants est vitale, il ne faut pas pour autant oublier de sécuriser les composants eux-mêmes. Ainsi, lors de la révision du code d'ASAX, il faudra impérativement veiller à la robustesse de l'implémentation. Il serait en effet fâcheux que l'IDS puisse être "planté", et donc neutralisé, à cause d'une erreur de programmation. De même, afin de limiter les possibilités de *buffer overflow*, on portera un soin particulier à la vérification de la taille de données avant de les manipuler.

6.4 Conclusion

Ce chapitre a montré que la sécurisation d'ASAX est indispensable si l'on veut utiliser l'IDS en situation réelle. Cependant, le choix d'un système cryptographique est une opération sensible. Les besoins des utilisateurs peuvent varier fortement selon le degré de protection souhaité ou encore selon les services d'authentification déjà présents sur le réseau.

Ainsi, il ne nous semble pas judicieux d'imposer à l'utilisateur un choix qui ne satisfera peut-être pas ses besoins. Il faudra donc veiller à ce que l'implémentation d'ASAX reste ouverte et puisse facilement intégrer un protocole de chiffrement et d'authentification des communications inter-composants.

Conclusions et perspectives de développement

Après avoir sensibilisé le lecteur à l'importance de la sécurité dans les réseaux informatiques, nous avons détaillé un outil de protection : le système de détection d'intrusion ASAX. Nous avons ensuite exposé les différents travaux que nous avons réalisés sur une version particulière de ce programme, à savoir "ASAX distribué".

Plus précisément, nous avons :

- porté ASAX distribué vers le système d'exploitation Solaris 2.7
- ajouté une dimension *network-based* à ASAX distribué en créant un adaptateur de format pour la source d'audit réseau TCPLogger
- créé un traceur d'événements en RUSSEL
- utilisé ce traceur pour développer quelques règles de détection
- intégré au langage RUSSEL des routines C de traitement d'adresses IP et de gestion de variables dynamiques

Nous pensons avoir mené à bien la tâche consistant au portage, l'adaptation et le déploiement d'ASAX distribué. Nous regrettons cependant de n'avoir pu développer une base de règles plus importante. En effet, le portage a mis en évidence un certain nombre de problèmes d'implémentation qu'il nous a fallu corriger.

L'utilisation pratique de l'IDS a également révélé un certain manque de souplesse ainsi que quelques problèmes plus structurels. Nous avons donc proposé une nouvelle architecture permettant de corriger les limitations rencontrées. Enfin, nous avons abordé la question de la sécurisation de l'IDS.

Si la version actuelle est fonctionnelle et efficace, elle demande par contre encore trop d'investissement de la part de l'utilisateur désirant la mettre en œuvre. Il est donc impérieux de réaliser une version plus "clef en main" de cet IDS si l'on veut que les utilisateurs potentiels s'y intéressent.

Les propositions abordées lors des deux derniers chapitres se veulent donc être les premiers pas menants à une nouvelle version d'ASAX distribué plus facile à déployer et à utiliser. Le chemin menant à l'IDS idéal est cependant encore long et certains points devront être approfondis. Citons par exemple :

- La création d'une "distribution ASAX" facilement installable et contenant des adaptateurs de format pour les sources d'audit les plus répandues. Comme nous l'avons évoqué au chapitre 3, une utilisation adéquate des outils `automake` et `autoconf` permettrait d'atteindre ce but.

- L'intégration d'un ensemble de scripts permettant à l'administrateur de gérer de manière confortable et transparente les éléments externes à ASAX, mais ayant toutefois un rôle important dans l'IDS. Citons, par exemple, l'élaboration de scripts controlant et configurant les différentes sources d'audit, ou encore, offrant des fonctionnalités de gestion de stockage et d'archivage des fichiers d'audit. Ces éléments sont fortement dépendants de l'environnement d'exécution de l'IDS (système d'exploitation, politique générale de gestion des archives, ...). Il nous semble donc opportun d'externaliser leur gestion afin de conserver le caractère universel d'ASAX.
- Le développement d'une base de règles de détection. Les IDS *knowledge-based* sont souvent jugés selon le nombre d'attaques qu'ils sont capables de détecter.
- Le développement d'une interface graphique. Lorsqu'on désire mettre en place un réseau d'analyse contenant de nombreux évaluateurs organisés en un arbre à plusieurs niveaux, la conception du fichier de description d'environnement (`edf`) peut vite se transformer en casse-tête. Il serait alors utile de disposer d'une interface graphique. Celle-ci permettrait littéralement de dessiner le graphe représentant le réseau d'analyse. De même, une représentation graphique des alarmes déclenchées peut alléger la charge cognitive de l'administrateur.
- Etudier les différentes stratégies d'utilisation et de déploiement d'ASAX. Est-il toujours pertinent d'utiliser ASAX distribué en dehors de l'analyse en temps-réel d'un réseau ? Si l'on désire lancer une analyse différée ou "post-mortem", l'utilisation de la version monoposte d'ASAX n'est-elle pas suffisante ? Et si tel est le cas, le déploiement dans un réseau peut-il être aisément simulé par un évaluateur d'une version monoposte ? Il serait donc utile de passer en revue les différents besoins des utilisateurs afin de pouvoir leur proposer une architecture et un déploiement adéquat d'ASAX, distribué ou monoposte.
- L'utilisation d'un langage générant du RUSSEL tel que LaDAA ou Sutekh faciliterait grandement l'élaboration de règles de détection. Quand une attaque possède plusieurs variantes, elles peuvent souvent être détectées par un même principe d'analyse. Toutefois, il est nécessaire d'écrire un ensemble de règles pour chacune de ces variantes. Dès lors, la rédaction de ces règles peut rapidement devenir fastidieuse pour l'administrateur qui, par conscience professionnelle, désire protéger son réseau.

Nous espérons sincèrement que le développement d'ASAX distribué se poursuivra dans un futur proche et permettra à cet IDS de connaître enfin le succès qu'il mérite.

Bibliographie

[Gib01] Steve Gibson, *The Strange Tale of the Denial of Service Attacks Against GRC.COM*, Tech. report, Gibson Research Corporation, juillet 2001, http ://grc.com/dos/grcdos.htm.

[Gér98] Françoise Gérard, *Définition et Implémentation d'un Langage Déclaratif pour l'Analyse d'Audit Trails*, Master's thesis, Institut d'Informatique, Facultés Universitaires Notre-Dame de la Paix Namur, Belgique, 1998.

[Gug00] Connie Guglielmo, *The ABCs of a DOS Attack*, Interactive Week, février 2000, http ://www.zdnet.com/intweek/stories/news/0,4164,2435990,00.html.

[HLCM94] Naji Habra, Baudouin Le Charlier, and Abdelaziz Mounji, *Advanced Security Audit Trail Analysis on Unix (ASAX also called SAT-X). Implementation Design of the NADF Evaluator*, Tech. report, Institut d'Informatique, Facultés Universitaires Notre-Dame de la Paix Namur, Belgique, septembre 1994.

[HLCMM92] Naji Habra, Baudouin Le Charlier, Abdelaziz Mounji, and Isabelle Mathieu, *ASAX : Software Architecture and Rule-Based Language for Universal Audit Trail Analysis*, European Symposium on Research in Computer Security (ESORICS'92) (Toulouse, France), Springer-Verlag, novembre 1992.

[HLCMM94] ———, *Preliminary Report on Advanced Security Audit Trail Analysis on Unix (ASAX also called SAT-X)*, Tech. report, Institut d'Informatique, Facultés Universitaires Notre-Dame de la Paix Namur, Belgique, décembre 1994.

[Jar] *Le jargon du Hacker*, http ://www.tuxedo.org/~esr/jargon/jargon.html.

[Ken99] Kristopher Kendall, *A Database of Computer Attacks for the Evaluation of Intrusion Detection Systems*, Master's thesis, Massachusetts Institute of Technology, juin 1999.

[KNMH00] Josué Kuri, Gonzalo Navarro, Ludovic Mé, and Laurent Heye, *A Pattern Matching Based Filter for Audit Reduction and Fast Detection of Potential Intrusions*, Recent Advances in Intrusion Detection (RAID 2000) (Toulouse, France) (Hervé Debar, Ludovic Mé, and S. Felix Wu, eds.), LNCS 1907, Springer-Verlag, octobre 2000, pp. 17–27.

[LHF+00] Richard Lippman, Joshua W. Haines, David J. Fried, Jonathan Korba, and Kumar Das, *Analysis and Results of the 1999 DARPA Off-Line Intrusion Detection Evaluation*, Recent Advances in Intrusion Detection (RAID 2000) (Toulouse, France) (Hervé Debar, Ludovic Mé, and S. Felix Wu, eds.), LNCS 1907, Springer-Verlag, octobre 2000, pp. 162–182.

[ME98] David MacKenzie and Ben Elliston, *Creating Automatic Configuration Scripts for Autoconf version 2.13*, GNU, edition 2.13 ed., décembre 1998, http ://www.gnu.org/manual/autoconf/html_mono/autoconf.html.

[MLCM93] Abdelaziz Mounji, Baudouin Le Charlier, and Isabelle Mathieu, *ASAX : Advanced Sequential File Analysis product*, novembre 1993.

[MLCZH94] Abdelaziz Mounji, Baudouin Le Charlier, Denis Zampunieris, and Naji Habra, *Preliminary Report on Distributed ASAX*, Tech. report, Institut d'Informatique, Facultés Universitaires Notre-Dame de la Paix Namur, Belgique, mai 1994.

[Mou97] Abdelaziz Mounji., *Languages and Tools for Rule-Based Distributed Intrusion Detection*, Ph.D. thesis, Institut d'Informatique, Facultés Universitaires Notre-Dame de la Paix Namur, Belgique, septembre 1997.

[MT98] David MacKenzie and Tom Tromey, *GNU Automake For version 1.3*, GNU, avril 1998, http ://www.gnu.org/manual/automake/html_mono/automake.html.

[NET94] Texas A & M University, CIS Network Group, *TAMU Security Tools : Netlog*, janvier 1994, http ://www.net.tamu.edu/network/tools/netlog.html.

[One96] Aleph One, *Smashing The Stack For Fun And Profit*, Phrack **7** (1996), no. 49, http ://www.phrack.org/show.php ?p=49&a=14.

[PD01] Jean-Philippe Pouzol and Mireille Ducassé, *From Declarative Signatures to Misuse IDS*, IRISA/INSA de Rennes, 2001.

[Ray01] Eric Steven Raymond, *How To Become A Hacker*, Thyrsus Enterprises, mars 2001, http ://www.tuxedo.org/~esr/faqs/hacker-howto.html.

[SAN] SANS Institute, *Intrusion Detection FAQ*, http ://www.sans.org/newlook/resources/IDFAQ/ID_FAQ.htm.

[SEN] *Site web du démon sendmail*, http ://www.sendmail.org.

[SNO] *Site web de SNORT*, http ://www.snort.org.

[Sun93] SunSoft, *Solaris SHIELD Basic Security Module*, octobre 1993.

[TRU] *Truss Man Page*, http ://toons.cs.ucsb.edu :4080/man ?truss.

[Vac] *FAQ. Almost Everything You Ever Wanted To Know About Security*, http ://www.vacia.is.tohoku.ac.jp/~s-yamane/articles/hacker/comp-security-FAQ/.

Annexe A

Codes RUSSEL

A.1 Code du traceur

```
rule catch_track;
var s : string;
begin
        if (strToInt(header32_event_ID) = 23) and (match('^..*/track$',path_0_path) = 1)
        -> begin
                msg_maz;
                msg_ajout('—Track detected— ',gettime(header32_date));
                msg_rapport(4);
                println('—Track detected— ',gettime(header32_date));
                trigger off for_next catch_vfork(strToInt(subject32_process_ID))
           end
        fi;
        trigger off for_next catch_track
end;


rule catch_vfork(pid :integer);
begin
        if (strToInt(subject32_process_ID) = pid) and (strToInt(header32_event_ID) = 25)
        -> trigger off for_next catch_fork(strToInt(arg32_0_argument_value));
        true -> trigger off for_next catch_vfork(pid)
        fi
end;


rule catch_fork(pid :integer);
begin
        if (strToInt(subject32_process_ID) = pid) and (strToInt(header32_event_ID) = 2)
        -> trigger off for_next spy(strToInt(arg32_0_argument_value));
        true -> trigger off for_next catch_fork(pid)
        fi
end;


rule spy(pid :integer);
begin
        if (strToInt(subject32_process_ID) = pid)
        -> begin
                .
                .
```

```
        /* Collecte et affichage d'informations */
                        .
                        .
                if (strToInt(header32_event_ID) = 25) or (strToInt(header32_event_ID) = 2)
                -> trigger off for_next spy(strToInt(arg32_0_argument_value))
                fi;

                if not (strToInt(header32_event_ID) = 1)
                -> trigger off for_next spy(pid)
                fi;

                msg_rapport(4)
            end;

        true-> trigger off for_next spy(pid)
        fi
end;

init_action;
begin
        trigger off for_next catch_track
end.
```

A.2 Sources des règles RUSSEL développées dans le cadre de l'élaboration d'une base de détection

Dans cette section sont présentés les différents codes des règles RUSSEL que nous avons élaborées. Afin de comprendre ces codes, il convient de définir les rôles des différentes routines qui nous on été fournies à l'INSA et que nous avons annexées au langage RUSSEL :

- nom_machine : Cette fonction donne le nom d'un hôte à partr de son adresse IP
- msg_maz : Cette fonction crée un rapport vide
- msg_ajout : Cette fonction ajoute une entrée au rapport créé
- msg_rapport : Cette fonction enregistre le rapport dans un fichier
- tuer_processus : Cette fonction tue la fonction correspondant à un PID fourni

A.2.1 Copie du fichier */etc/passwd*

La règle `open_source` détecte les ouvertures en mode lecture du fichier `/etc/passwd`. Si une occurence apparaît, elle retient le numéro de l'utilisateur qu'elle utilise comme paramètre pour déclencher la règle `open_dest`. `open_source` se déclenche continuellement elle-même car elle détecte le premier événement au sein de la signature.

```
rule open_source;
begin
        if (strToInt(header32_event_ID) = 72) and (match('.*/etc/passwd$',path_0_path)=1)
        -> trigger off for_next open_dest(strToInt(subject32_user_ID))
        fi;
        trigger off for_next open_source
end;
```

La règle `open_dest` détecte l'ouverture en mode écriture d'un fichier par le même utilisateur que celui qui a ouvert le fichier `/etc/passwd`. Si elle trouve une occurence de cet événement, elle déclenche la règle `read` avec l'utilisateur et le nom du fichier destination comme paramètres.

```
rule open_dest(user :integer);
begin
        if
            ((strToInt(header32_event_ID) = 78) or (strToInt(header32_event_ID) = 79))
                        and (strToInt(subject32_user_ID) = user)
                -> trigger off for_next read(user,path_0_path)


            true -> trigger off for_next open_dest(user)
        fi
end;
```

La règle `read` cherche après une lecture du fichier `/etc/passwd`. Si elle en trouve une faite par l'utilisateur `user`, elle déclenche la règle `write` avec l'utilisateur et le fichier de destination comme paramètres.

```
rule read(user :integer; file_dest :string);
begin
        if
            (strToInt(header32_event_ID) = 192) and (strToInt(subject32_user_ID) = user)
                -> begin
                        if (match('^..*/etc/passwd$',path_0_path) = 1)
                        -> trigger off for_next write(user,file_dest)
                        fi
                   end;


            true -> trigger off for_next read(user,file_dest)
        fi
end;
```

La règle `write` cherche après une écriture dans le fichier destination. Si elle en trouve une faite par l'utilisateur `user`, alors il faut lever une alarme. On ne peut garantir que ce qui est écrit est ce qui a été lu par manque d'informations fournies par BSM.

```
rule write(user :integer; file_dest :string);
begin
        if
            (strToInt(header32_event_ID) = 195) and (path_0_path = file_dest)
                and (strToInt(subject32_user_ID) = user)
                -> trigger off for_current alarme(user,file_dest);


            true -> trigger off for_next write(user,file_dest)
        fi
end;

rule alarme(user :integer; file_dest :string);
begin
        println(user,' a copie /etc/passwd dans ',file_dest);
        msg_maz;
        msg_ajout('COPIE PASSWD :');
        msg_ajout(' machine=',nom_machine(strToInt(subject32_machine_ID)));
```

```
        msg_ajout(' utilisateur=',user);
        msg_ajout(' copie_fichier=',file_dest);
        msg_ajout(' date=',gettime(header32_date));
        msg_rapport(1)
end;

init_action;
begin
        trigger off for_next open_source
end.
```

A.2.2 Vol de shell

La règle `open_source` détecte les ouvertures en mode lecture du fichier `/bin/sh`. A toute occurence détectée, elle retient le numéro de l'utilisateurqu'elle utilise comme paramètre pour déclencher la règle `open_dest`. `opne_source` se déclenche continuellement elle-même car elle détecte le premier événement au sein de la signature.

```
rule open_source;
begin
        if (strToInt(header32_event_ID) = 72) and (match('^..*/bin/sh$',path_0_path) = 1)
        -> trigger off for_next open_dest(strToInt(subject32_user_ID))
        fi;


        trigger off for_next open_source
end;
```

La règle `open_dest` détecte l'ouverture en mode écriture d'un fichier par le même utilisateur que celui qui a ouvert le fichier `/bin/sh`. Si elle trouve une occurence de cet événement, elle déclenche la règle `read` avec l'utilisateur et le nom du fichier destination comme paramètres.

```
rule open_dest(user :integer);
begin
        if
            ((strToInt(header32_event_ID) = 78) or (strToInt(header32_event_ID) = 79))
                        and (strToInt(subject32_user_ID) = user)
                -> trigger off for_next read(user,path_0_path)


            true -> trigger off for_next open_dest(user)
        fi
end;
```

La règle `read` cherche après une lecture du fichier `/bin/sh`. Si elle en trouve une faite par l'utilisateur `user`, elle déclenche la règle `write` avec l'utilisateur et le nom du fichier de destination comme paramètres.

```
rule read(user :integer; file_dest :string);
begin
        if
            (strToInt(header32_event_ID) = 192) and (strToInt(subject32_user_ID) = user)
                -> begin
                        if (match('^..*/bin/sh$',path_0_path) = 1)
                        -> trigger off for_next write(user,file_dest)
```

```
                    fi
                end;

        true -> trigger off for_next read(user,file_dest)
    fi
end;
```

La règle `write` cherche après une écriture dans le fichier destination. Si elle en trouve une faite par l'utilisateur `user`, alors on peut passer à la deuxième étape de la signature, c'est à dire la détection du `chmod 4755`. On va donc déclencher la règle `chmod` avec le nom du fichier destination et l'utilisateur comme paramètres.

```
rule write(user :integer; file_dest :string);
begin
    if
        (strToInt(header32_event_ID) = 195) and (path_0_path = file_dest)
            and (strToInt(subject32_user_ID) = user)
            -> trigger off for_next chmod(user,file_dest)

        true -> trigger off for_next write(user,file_dest)
    fi
end;
```

La règle `chmod` détecte un `chmod 4755` sur le fichier `file_dest`. Si elle trouve une occurence de cet événement, elle lève une alarme.

```
rule chmod(user :integer; file_dest :string);
begin
    if
        (strToInt(header32_event_ID) = 10)
            and (strToInt(arg32_0_argument_value) = 2541)
            and (path_0_path = file_dest)
            -> trigger off for_current alarme(user,file_dest);

        true -> trigger off for_next chmod(user,file_dest)
    fi
end;

rule alarme(user :integer; file_dest :string);
begin
    println('L'utilisateur ',user,' a copie /bin/sh dans ',file_dest
                    ,'et un chmod 4755 a ete execute sur la copie');
    msg_maz;
    msg_ajout('VOL DE SHELL :');
    msg_ajout(' machine=',nom_machine(strToInt(subject32_machine_ID)));
    msg_ajout(' utilisateur=',user);
    msg_ajout(' copie_shell=',file_dest);
    msg_ajout(' date=',gettime(header32_date));
    msg_rapport(1)
end;

init_action;
begin
    trigger off for_next open_source
end.
```

A.2.3 Attaque sur le démon *sendmail*

La règle `open_source` détecte les ouvertures en mode lecture du fichier `/bin/sh`. Si une occurence apparaît, elle retient le numéro de l'utilisateur qu'elle utilise comme paramètre pour déclencher la règle `open_dest`. La règle `open_source` se déclenche continuellement elle-même car elle détecte le premier événement au sein de la signature.

```
rule open_source;
begin
        if (strToInt(header32_event_ID) = 72) and (match('^..*/bin/sh$',path_0_path) = 1)
        -> trigger off for_next open_dest(strToInt(subject32_user_ID))
        fi;


        trigger off for_next open_source
end;
```

La règle `open_dest` détecte l'ouverture en mode écriture du fichier `/var/mail/root` par le même utilisateur que celui qui a ouvert le fichier `/bin/sh`. Si elle trouve une occurence de cet événement, elle déclenche la règle `read` avec l'utilisateur et le nom du fichier destination comme paramètres.

```
rule open_dest(user :integer);
begin
        if
            ((strToInt(header32_event_ID) = 78) or (strToInt(header32_event_ID) = 79))
                and (strToInt(subject32_user_ID) = user)
                and(match('^..*/var/mail/root$',path_0_path) = 1)
                -> trigger off for_next read(user);


            true -> trigger off for_next open_dest(user)
        fi
end;
```

La règle read cherche après une lecture du fichier /bin/sh. Si elle en trouve une faite par l'utilisateur user, alors elle déclenche la règle write avec l'utilisateur, les descripteurs de fichiers et le fichier de destination comme paramètres.

```
rule read(user :integer);
begin
        if
            (strToInt(header32_event_ID) = 192) and (strToInt(subject32_user_ID) = user)
                and (match('^..*/bin/sh$',path_0_path) = 1)
                -> trigger off for_next write(user);


            true -> trigger off for_next read(user)
        fi
end;
```

La règle `write` cherche après une écriture dans le fichier `/var/mail/root`. Si elle en trouve une faite par l'utilisateur `user`, on peut passer à la deuxième étape de la signature, c'est à dire la détection du `chmod 4755`. On va donc déclencher la règle `chmod` avec l'utilisateur comme paramètre.

```
rule write(user :integer);
begin
        if
            (strToInt(header32_event_ID) = 195)
                and (match('^..*/var/mail/root$',path_0_path) = 1)
                and(strToInt(subject32_user_ID) = user)
                -> trigger off for_next chmod(user)


            true -> trigger off for_next write(user)
        fi
end;
```

La règle `chmod` détecte un `chmod 4755` sur le fichier `/var/mail/root`. Si elle le trouve, on peut passer à la détection de la création d'un fichier vide. La règle `touch` est donc déclenchée.

```
rule chmod(user :integer);
begin
        if
            (strToInt(header32_event_ID) = 10) and (strToInt(subject32_user_ID) = user)
                and (strToInt(arg32_0_argument_value) = 2541)
                and (match('^..*/var/mail/root$',path_0_path) = 1)
                -> trigger off for_next touch;


            true-> trigger off for_next chmod(user)
        fi
end;
```

La règle `touch` détecte la création d'un fichier vide. Si la détection est fructueuse, on peut continuer la détection en recherchant l'exécution de `sendmail`. Le chemin d'accès au fichier créé est passé à la règle `mail`.

```
rule touch;
begin
        if
            (strToInt(header32_event_ID) = 4)
                -> trigger off for_next mail(path_0_path)


            true -> trigger off for_next touch
        fi
end;
```

La règle `mail` détecte l'exécution du programme "mail" et si celui-ci est destiné à envoyer un message à l'administrateur `root`. Si elle trouve une occurence de cet événement, il faut vérifier si elle le message envoyé est bien le fichier `file` et ce, en déclanchant la règle `send_mail` et en lui passant le fichier en question et le PID du processus lancé de `sendmail` comme paramètres.

```
rule mail(file :string);
begin
        if
            (strToInt(header32_event_ID) = 23)
                and (match('^..*/mail$',path_0_path) = 1)
                and (match('root',getarg(exec_args_env_args,1)) = 1)
                -> trigger off for_next send_mail(file,strToInt(subject32_process_ID));
```

```
            true -> trigger off for_next mail(file)
        fi
end;
```

La règle `send_mail` détecte si le programme `mail` lancé lit le fichier vide `file` créé auparavant. Si une occurence de ce événement apparaît, il faut lever une alarme.

```
rule send_mail(file :string; pid :integer);
begin
        if
            (strToInt(header32_event_ID) = 192)
                and (strToInt(subject32_process_ID) = pid)
                and (path_0_path = file)
                trigger off for_current alarme;


            true -> trigger off for_next send_mail(file,pid)
        fi
end;

rule alarme;
begin
        println('-----------------------------------');
        println(gettime(header32_date));
        println(strToInt(subject32_user_ID), ' is trying to get a root shell via sendmail');
        println('-----------------------------------');
        msg_maz;
        msg_ajout('SENDMAIL EXPLOITATION :');
        msg_ajout(' machine=',nom_machine(strToInt(subject32_machine_ID)));
        msg_ajout(' utilisateur=',strToInt(subject32_user_ID));
        msg_ajout(' date=',gettime(header32_date));
        msg_rapport(1)
end;

init_action;
begin
        trigger off for_next open_source
end.
```

A.2.4 Attaque *user-to-root* sur le programme *eject*

La règle `exec_eject` détecte l'exécution du programme `eject` par une autre personne que `root`. Si une occurence apparaît alors il faut vérifier si le *user-id* de l'utilisateur change via la règle `set_uid`. On passe à cette dernière le PID du programme `eject` qui a été lancé.

```
rule exec_eject;
begin
        if (strToInt(header32_event_ID) = 23)
                and (match('^..*/usr/bin/eject$',path_0_path) = 1)
                and not (strToInt(subject32_real_user_ID) = 0)
                -> trigger off for_next set_uid(strToInt(subject32_process_ID))
        fi;


        trigger off for_next exec_eject
end;
```

La règle `set_uid` détecte si l'utilisateur devient `root` au sein du programme `eject`. Si le cas apparaît, il faut alors détecter l'exécution d'un shell lancée à partir du programme `eject` lancé. Comme le user-id est passé à `root`, l'utilisateur aura tous les privilèges dans ce nouveau shell.

```
rule set_uid(pid :integer);
begin
        if
            (strToInt(header32_event_ID) = 200)
                and (strToInt(subject32_process_ID) = pid)
                and (strToInt(arg32_0_argument_value) = 0)
                -> trigger off for_next exec_sh(pid);


            true -> trigger off for_next set_uid(pid)
        fi
end;
```

La règle `exec_sh` détecte l'exécution d'un shell avec privilèges de `root` au sein de l'exécution du programme `eject`. Si le cas se présente, il faut lever une alarme.

```
rule exec_sh(pid :integer);
begin
        if
            (strToInt(header32_event_ID) = 23)
                and (match('^..*/bin/sh$',path_0_path) = 1)
                and (strToInt(subject32_real_user_ID) = 0)
                -> trigger off for_current alarme;


            true -> trigger off for_next exec_sh(pid)
        fi
end;

rule alarme;
begin
        tuer_processus(strToInt(subject32_process_ID));
        println('------------------------------------');
        println('User 2 Root Attack detected!!!');
        println('Attack : Buffer overflow on /usr/bin/eject');
        println('Date : ',gettime(header32_date));
        println('User ID : ',strToInt(subject32_audit_ID));
        println('Host IP : ',getIP(strToInt(subject32_machine_ID)));
        println('Host Name : ',nom_machine(strToInt(subject32_machine_ID)));
        println('------------------------------------');


        msg_maz;
        msg_ajout('USER 2 ROOT - EJECT BUFFER OVERFLOW :');
        msg_ajout(' machine=',nom_machine(strToInt(subject32_machine_ID)));
        msg_ajout(' ip_machine=',getIP(strToInt(subject32_machine_ID)));
        msg_ajout(' utilisateur=',strToInt(subject32_audit_ID));
        msg_ajout(' date=',gettime(header32_date));
        msg_rapport(1)
end;

init_action;
begin
        trigger off for_next exec_eject
end.
```

A.2.5 Tentatives de connexion à des chevaux de Troie

```
global nb : integer;


rule show_portscan(trojan_name :string);
begin
        println('**** Port Scan on port number ',strToInt(TCPL_tcpdstport),
                                                    '. Possible Trojan : ',trojan_name,' ****');
        println('Timestamp : ',gettime(TCPL_seconds));
        println('IP Source : ',getIP(strToInt(TCPL_ipsrcaddr)));
        println('Machine Source : ',nom_machine(strToInt(TCPL_ipsrcaddr)));
        println('IP Destination : ',getIP(strToInt(TCPL_ipdstaddr)));
        println('Machine Destination : ',nom_machine(strToInt(TCPL_ipdstaddr)));
        println('TCP Source Port : ',strToInt(TCPL_tcpsrcport));
        println('TCP Destination Port : ',strToInt(TCPL_tcpdstport));
        println('');


        msg_maz;
        msg_ajout('PORT SUSPECT :');
        msg_ajout(' machine_source=',nom_machine(strToInt(TCPL_ipsrcaddr)));
        msg_ajout(' IP_machine_source=',getIP(strToInt(TCPL_ipsrcaddr)));
        msg_ajout(' machine_destination=',nom_machine(strToInt(TCPL_ipdstaddr)));
        msg_ajout(' IP_machine_destination=',getIP(strToInt(TCPL_ipdstaddr)));
        msg_ajout(' TCP_Source_Port=',strToInt(TCPL_tcpsrcport));
        msg_ajout(' TCP_Destination_Port=',strToInt(TCPL_tcpdstport));
        msg_ajout(' date=',gettime(TCPL_seconds));
        msg_rapport(2);


        nb := nb+1
end;

rule detect_portscan;
begin
        if present TCPL_tcpsrcport
        -> begin
                if
                    (strToInt(TCPL_tcpdstport) = 456)
                        -> trigger off for_current show_portscan('Hackers Paradise');
                    (strToInt(TCPL_tcpdstport) = 555)
                        -> trigger off for_current show_portscan('Ini-Killer, Phase Zero
                                                                  or Stealth Spy');
                    (strToInt(TCPL_tcpdstport) = 666)
                        -> trigger off for_current show_portscan('Satanz Backdoor');
                    (strToInt(TCPL_tcpdstport) = 1001)
                        -> trigger off for_current show_portscan('Silencer or WebEx');
                    (strToInt(TCPL_tcpdstport) = 1011)
                        -> trigger off for_current show_portscan('Doly Trojan');
                    (strToInt(TCPL_tcpdstport) = 1170)
                        -> trigger off for_current show_portscan('Psyber Stream Server
                                                                  or Voice');
                    (strToInt(TCPL_tcpdstport) = 1234)
                        -> trigger off for_current show_portscan('Ultors Trojan');
                    (strToInt(TCPL_tcpdstport) = 1245)
                        -> trigger off for_current show_portscan('VooDoo Doll');
                    (strToInt(TCPL_tcpdstport) = 1492)
                        -> trigger off for_current show_portscan('FTP99CMP');
                    (strToInt(TCPL_tcpdstport) = 1600)
```

```
    -> trigger off for_current show_portscan('Shivka-Burka');
(strToInt(TCPL_tcpdstport) = 1807)
    -> trigger off for_current show_portscan('SpySender');
(strToInt(TCPL_tcpdstport) = 1981)
    -> trigger off for_current show_portscan('Shockrave');
(strToInt(TCPL_tcpdstport) = 1999)
    -> trigger off for_current show_portscan('BackDoor');
(strToInt(TCPL_tcpdstport) = 2001)
    -> trigger off for_current show_portscan('Trojan Cow');
(strToInt(TCPL_tcpdstport) = 2023)
    -> trigger off for_current show_portscan('Ripper');
(strToInt(TCPL_tcpdstport) = 2115)
    -> trigger off for_current show_portscan('Bugs');
(strToInt(TCPL_tcpdstport) = 2140)
    -> trigger off for_current show_portscan('Deep Throat
                                              or The Invasor');
(strToInt(TCPL_tcpdstport) = 2801)
    -> trigger off for_current show_portscan('Phineas Phucker');
(strToInt(TCPL_tcpdstport) = 3024)
    -> trigger off for_current show_portscan('WinCrash');
(strToInt(TCPL_tcpdstport) = 3129)
    -> trigger off for_current show_portscan('Masters Paradise');
(strToInt(TCPL_tcpdstport) = 3150)
    -> trigger off for_current show_portscan('Deep Throat
                                              or The Invasor');
(strToInt(TCPL_tcpdstport) = 3700)
    -> trigger off for_current show_portscan('Portal of Doom');
(strToInt(TCPL_tcpdstport) = 4092)
    -> trigger off for_current show_portscan('WinCrash');
(strToInt(TCPL_tcpdstport) = 4590)
    -> trigger off for_current show_portscan('ICQTrojan');
(strToInt(TCPL_tcpdstport) = 5000)
    -> trigger off for_current show_portscan('Sockets de Troie');
(strToInt(TCPL_tcpdstport) = 5001)
    -> trigger off for_current show_portscan('Sockets de Troie');
(strToInt(TCPL_tcpdstport) = 5321)
    -> trigger off for_current show_portscan('Firehotcker');
(strToInt(TCPL_tcpdstport) = 5400)
    -> trigger off for_current show_portscan('Blade Runner');
(strToInt(TCPL_tcpdstport) = 5401)
    -> trigger off for_current show_portscan('Blade Runner');
(strToInt(TCPL_tcpdstport) = 5402)
    -> trigger off for_current show_portscan('Blade Runner');
(strToInt(TCPL_tcpdstport) = 5569)
    -> trigger off for_current show_portscan('Robo-Hack');
(strToInt(TCPL_tcpdstport) = 5742)
    -> trigger off for_current show_portscan('WinCrash');
(strToInt(TCPL_tcpdstport) = 6670)
    -> trigger off for_current show_portscan('DeepThroat');
(strToInt(TCPL_tcpdstport) = 6671)
    -> trigger off for_current show_portscan('DeepThroat');
(strToInt(TCPL_tcpdstport) = 6969)
    -> trigger off for_current show_portscan('GateCrasher
                                              or Priority');
(strToInt(TCPL_tcpdstport) = 7000)
    -> trigger off for_current show_portscan('Remote Grab');
(strToInt(TCPL_tcpdstport) = 7300)
    -> trigger off for_current show_portscan('NetMonitor');
```

```
(strToInt(TCPL_tcpdstport) = 7301)
    -> trigger off for_current show_portscan('NetMonitor');
(strToInt(TCPL_tcpdstport) = 7306)
    -> trigger off for_current show_portscan('NetMonitor');
(strToInt(TCPL_tcpdstport) = 7307)
    -> trigger off for_current show_portscan('NetMonitor');
(strToInt(TCPL_tcpdstport) = 7308)
    -> trigger off for_current show_portscan('NetMonitor');
(strToInt(TCPL_tcpdstport) = 7789)
    -> trigger off for_current show_portscan('ICKiller');
(strToInt(TCPL_tcpdstport) = 9872)
    -> trigger off for_current show_portscan('Portal of Doom');
(strToInt(TCPL_tcpdstport) = 9873)
    -> trigger off for_current show_portscan('Portal of Doom');
(strToInt(TCPL_tcpdstport) = 9874)
    -> trigger off for_current show_portscan('Portal of Doom');
(strToInt(TCPL_tcpdstport) = 9875)
    -> trigger off for_current show_portscan('Portal of Doom');
(strToInt(TCPL_tcpdstport) = 9989)
    -> trigger off for_current show_portscan('iNi-Killer');
(strToInt(TCPL_tcpdstport) = 10067)
    -> trigger off for_current show_portscan('Portal of Doom');
(strToInt(TCPL_tcpdstport) = 10167)
    -> trigger off for_current show_portscan('Portal of Doom');
(strToInt(TCPL_tcpdstport) = 11000)
    -> trigger off for_current show_portscan('Senna Spy');
(strToInt(TCPL_tcpdstport) = 11223)
    -> trigger off for_current show_portscan('Progenic trojan');
(strToInt(TCPL_tcpdstport) = 12223)
    -> trigger off for_current show_portscan('Hack99 KeyLogger');
(strToInt(TCPL_tcpdstport) = 12345)
    -> trigger off for_current show_portscan('GabanBus or NetBus');
(strToInt(TCPL_tcpdstport) = 12346)
    -> trigger off for_current show_portscan('GabanBus or NetBus');
(strToInt(TCPL_tcpdstport) = 12361)
    -> trigger off for_current show_portscan('Whack-a-mole');
(strToInt(TCPL_tcpdstport) = 12362)
    -> trigger off for_current show_portscan('Whack-a-mole');
(strToInt(TCPL_tcpdstport) = 16969)
    -> trigger off for_current show_portscan('Priority');
(strToInt(TCPL_tcpdstport) = 20001)
    -> trigger off for_current show_portscan('Millennium');
(strToInt(TCPL_tcpdstport) = 20034)
    -> trigger off for_current show_portscan('NetBus 2 Pro');
(strToInt(TCPL_tcpdstport) = 21544)
    -> trigger off for_current show_portscan('GirlFriend');
(strToInt(TCPL_tcpdstport) = 22222)
    -> trigger off for_current show_portscan('Prosiak');
(strToInt(TCPL_tcpdstport) = 23456)
    -> trigger off for_current show_portscan('Evil FTP
                                    or Ugly FTP');
(strToInt(TCPL_tcpdstport) = 26274)
    -> trigger off for_current show_portscan('Delta');
(strToInt(TCPL_tcpdstport) = 31337)
    -> trigger off for_current show_portscan('Back Orifice');
(strToInt(TCPL_tcpdstport) = 31338)
    -> trigger off for_current show_portscan('Back Orifice
                                    or DeepBO');
```

```
                (strToInt(TCPL_tcpdstport) = 31339)
                    -> trigger off for_current show_portscan('NetSpy DK');
                (strToInt(TCPL_tcpdstport) = 31666)
                    -> trigger off for_current show_portscan('BOWhack');
                (strToInt(TCPL_tcpdstport) = 33333)
                    -> trigger off for_current show_portscan('Prosiak');
                (strToInt(TCPL_tcpdstport) = 34324)
                    -> trigger off for_current show_portscan('Biggluck or TN');
                (strToInt(TCPL_tcpdstport) = 40412)
                    -> trigger off for_current show_portscan('The Spy');
                (strToInt(TCPL_tcpdstport) = 40421)
                    -> trigger off for_current show_portscan('Masters Paradise');
                (strToInt(TCPL_tcpdstport) = 40422)
                    -> trigger off for_current show_portscan('Masters Paradise');
                (strToInt(TCPL_tcpdstport) = 40423)
                    -> trigger off for_current show_portscan('Masters Paradise');
                (strToInt(TCPL_tcpdstport) = 40426)
                    -> trigger off for_current show_portscan('Masters Paradise');
                (strToInt(TCPL_tcpdstport) = 47262)
                    -> trigger off for_current show_portscan('Delta');
                (strToInt(TCPL_tcpdstport) = 50505)
                    -> trigger off for_current show_portscan('Sockets de Troie');
                (strToInt(TCPL_tcpdstport) = 50766)
                    -> trigger off for_current show_portscan('Fore');
                (strToInt(TCPL_tcpdstport) = 53001)
                    -> trigger off for_current show_portscan('Remote Windows
                                                              Shutdown');
                (strToInt(TCPL_tcpdstport) = 61466)
                    -> trigger off for_current show_portscan('Telecommando');
                (strToInt(TCPL_tcpdstport) = 65000)
                    -> trigger off for_current show_portscan('Devil')
            fi
        end
    fi;

    trigger off for_next detect_portscan
end;
```

Annexe B

Codes sources de modules annexés à RUSSEL

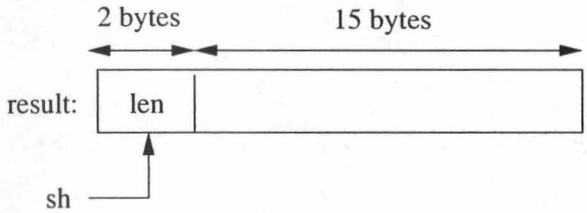
B.1 Code du module de traitement IP

This is a library which allows the user to manipulate IP addresses in RUSSEL. *The functions provided to RUSSEL are :* *- getIP : gets a "X.X.X.X" IP from an integer* *- IPtoInt : translates a "X.X.X.X" IP into an integer* *- inDomain : checks if an IP address belongs to a domain*

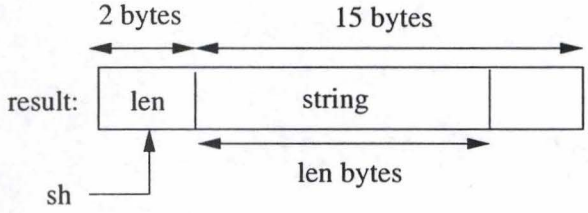
```
/*---copy this section and paste it in the ldf file-----
string getIP(integer :val) : iptools;
integer IPtoInt(string :val) : iptools;
integer inDomain(integer :val,integer :val,integer :val) : iptools;
-------------------------*/

#include <stdio.h>

char result[17]; // 15 characters + the "short" length (2) = 17
```

FUNCTION :	*char* create_RUSSEL_Str(char* s)*
Precondition :	*s is a NULL-terminated string*
Postcondidtion :	*s is not modified*
Return Value :	*result, the string s in RUSSEL-format*

```
char* create_RUSSEL_Str(char* s)
{
  unsigned short *sh;
  unsigned short len = strlen(s);
  sh = (unsigned short*) result;
  *sh = (unsigned short) len; // puts the length at the beginning of the RUSSEL string
```



strncpy(result+2,s,len) ; /* *puts the string after the length without the 0-terminal* */



return result ;
}

FUNCTION :	*char* getIP(unsigned int ip)*
Precondition :	*ip is an IP address in integer format*
Postcondition :	*see Return Value*
Return Value :	*ip in "X.X.X.X" format*

```
char* getIP(char* s)
{
  char ipaddr[16];
  unsigned int ip = *((unsigned int*) s); // The IP address is extracted from the
argument area
  sprintf(ipaddr,"%i.%i.%i.%i",ip≫24, (ip≪8)≫24, (ip≪16)≫24, (ip≪24)≫24); // The
IP is build in C-format (NULL-terminated)
  return create_RUSSEL_Str(ipaddr);
}
```

FUNCTION :	*int IPtoInt(char* ip)*
Precondition :	*ip is "X.X.X.X"-like IP address*
Postcondition :	*see return value*
Return Value :	*result, ip in integer format*

```
int IPtoInt(char* s)
{
  unsigned short** sh;
  unsigned short length;
  char* ip; // the ip in NULL-terminated format
  char* s1;
  char result[4]; // the ip in integer format
  int i = 0; // counter
  int j = 0; // counter : part number in an IP address "X.X.X.X" -> 0.1.2.3

  sh = (unsigned short**) s;
  length = **sh; // extracts the length of the RUSSEL string
```

```
  ip = (char*) malloc (length+1);
  s1 = (char*)(*sh+1); // extracts the string (IP) itself
  strncpy(ip,s1,length); // copies it into the variable 'ip'
  ip[length] = 0; // appends the 0-terminal to 'ip'

  /* An address IP is represented by a 32bits-integer. In fact, 4 X 8bits.
     Indeed, there are 4 parts separated by a dot in an IP address. So the
     section builds the 32bits(4X8)-integer one byte after the other */

  for (;;)
    {
      if (ip[i] == 0)
        {
          result[j] = (char) atoi(ip);
          break;
        }
      else if (ip[i] == '.')
        {
          ip[i] = 0;
          result[j] = (char) atoi(ip);
          ip = &ip[i+1];
          i = 0;
          j++;
        }
      else i++;
    }
  free(ip);
  return *((int*)result);
}
```

FUNCTION :	*int inDomain(int ip_src,int ip_dom,int mask)*
Precondition :	*- ip_src is the IP to be checked* *- ip_dom is the IP representing the Domain* *- mask is the IP mask to be applied during the test*
Postcondition :	*see return value*
Return Value :	*- 1 if ip_src belongs to the domain represented by ip_dom* *- 0 else*

```
int inDomain(char* s)
{
  int ip_src = *((int*) s);
  int ip_dom = *((int*) (s+4));
  int mask = *((int*) (s+8));

  int temp = ip_src & mask;
  if ((ip_dom & mask) == temp) return 1;
  else return 0;
}
```

B.2 Code du module de gestion de variables dynamiques

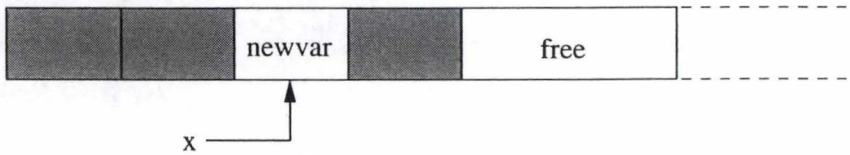
This is a library which allows the user to manipulate dynamic variables in RUSSEL.
The type of the variables is only integer (32-bits maximum)
The functions provided to RUSSEL are :
- *newvar : creates a new dynamic variable*
- *setvar : sets the value of a dynamic variable*
- *getvar : returns the value of a dynamic variable*
- *deletevar : wipes a dynamic variable from memory*
- *dec : decreases a dynamic variable by 1*
- *inc : increases a dynamic variable by 1*

```
/*----copy this section and paste it in the ldf file----
integer newvar(integer :val) : vardyn;
integer getvar(integer :val) : vardyn;
setvar(integer :val,integer :val) : vardyn;
deletevar(integer :val) : vardyn;
dec(integer :val) : vardyn;
inc(integer :val) : vardyn;
-------------------------*/

#include <stdlib.h>
#include <stdio.h>

#define BLOCK 2000

/* Overview
The variables are stocked in an array of integer. In the language C, you can
reference a variable with a memory address (int* for instance). Here, in RUSSEL,
you reference it with the index of the variable in the array.
For example, in RUSSEL :
var x : integer;
x := newvar(5);
The value of x is the index of the cell containing the dynamic variable.
So, to get the true value, you must use the primitive getvar(x)
All of these will be explained in details in the following */

int* buf = NULL; // The pointer referencing the array of integer
int size2; // The size of the array
char* pres; // Array of booleans which determinates if a cell of the array is occupied or
not
```

FUNCTION : *int getfree()*

Precondition : /

Postcondition : *The first free cell in the array is allocated.*



- *x is the index of the new allocated cell*
- *all the cells before x are allocated*
- *all the content of the array is not modified*
- *the array is allocated if it wasn't done yet*
- *the array is reallocated if there isn't any vacant cell*

Return Value :
- *x, the index of the new allocated variable*
- *-1 if a problem occured*

```
int getfree()
{
  int* buf2; // new array in case of reallocation
  char* pres2; // the new array of boolean in case of reallocation

  if (buf) // case where the array is allocated
    {
      int i;
      for (i = 0; i < size2; i++)
        if (!pres[i]) return i; // returns the index of the first free cell

      if ((buf2 = realloc (buf, ((size2+BLOCK) * sizeof(int)))) && (pres2 = realloc
(pres, (size2+BLOCK)))) // if the reallocation happened without any problem
        {
          buf = buf2;
          pres = pres2;
          size2 += BLOCK; // The size of the array has increased by BLOCK (2000)
          return ((size2 - BLOCK) + 1); // Indeed the return value is the index of the
first cell of the new allocated area
        }
      else
        {
          fprintf(stderr,"getfree : error realloc\n");
          return -1;
        }
    }
  else // case where the array doesn't exist yet
    {
      buf = (int*) malloc (BLOCK * sizeof(int));
      pres = (char*) malloc (BLOCK); // The arrays are allocated
      size2 = BLOCK;
      return 0; // returns the index of the first cell of the array
    }
}
```

FUNCTION : *int newvar(int val)*

Precondition : /
Postcondition : - The first free cell in the array is allocated (see function getfree)
- *x is the index of the cell allocated*
- *buf[x] = val*
- *buf[i] is not modified, $0 \leq i \leq size2$ and $i \neq x$*
- *pres[x] = 1*
- *pres[i] is not modified, $0 \leq i \leq size2$ and $i \neq x$*

Return Value : - *x, the index of the new allocated variable*
- *-1 if a problem occured*

```
int newvar (char* s)
{
  int result;
  int val = *((int*) s); // extracts val from the argument area
  result = getfree(); // allocates a cell of the array
  if (result >= 0) // if no error occured
    {
      buf[result] = val;
      pres[result] = 1; // updates the presence table (the cell number result is allocated
    }
  return result; // returns the index of the new allocated cell to RUSSEL
}
```

FUNCTION : *setvar(int id, int val)*

Precondition : /
Postcondition : - *if no error occured*
- *buf[id] = val*
- *buf[i] is not modified, $0 \leq i \leq size2$ and $i \neq id$*
- *if $id \geq size2$ -> error msg*
- *if buf[id] not allocated (!pres[id]) -> error msg*

```
void setvar (char* s)
{
  int id = *((int*) s);
  int val = *((int*) (s+4)); // extracts the values from the argument area

  if (id >= size2) fprintf(stderr,"setvar : variable out of bounds\n");
  else if ( !pres[id]) fprintf(stderr,"setvar : variable doesn't exist\n");
  else buf[id] = val;
}
```

FUNCTION : *int getvar(int id)*

Precondition : /
Postcondition : *The content of the array is not modified*
Return Value : - *buf[id] if no error occured*
- *0 if $id \geq size2$ or pres[id] = 0 (error)*

```
int getvar (char* s)
{
  int id = *((int*) s);
```

```
  if (id >= size2)
    {
      fprintf(stderr,"getvar : variable out of bounds. returns(0)\n");
      return 0;
    }
  else if (pres[id]) return buf[id];
  else
    {
      fprintf(stderr,"getvar : variable doesn't exist. returns(0)\n");
      return 0;
    }
}
```

FUNCTION :	*int deletevar(int id)*
Precondition :	*/*
Postcondition :	*- if id < size2*
	- pres[id] = 0
	- pres[i] is not modified, $0 \leq i \leq$ size2 and $i \neq$ id
	- pres is not modified else -> error msg

```
void deletevar (char* s)
{
  int id = *((int*) s);

  if (id >= size2) fprintf(stderr,"deletevar : variable out of bounds\n");
  else pres[id] = 0;
}
```

FUNCTION :	*dec(int id)*
Precondition :	*/*
Postcondition :	*- if id < size2 and pres[id] = 1*
	- buf[id] = buf[id] - 1
	- buf[i] is not modified, $0 \leq i \leq$ size2 and $i \neq$ id
	- buf is not modified else -> error msg

```
void dec (char* s)
{
  int id = *((int*) s);
  if (id >= size2) fprintf(stderr,"dec : variable out of bounds\n");
  else if (pres[id]) buf[id]--;
  else fprintf(stderr,"dec : variable doesn't exist\n");
}
```

FUNCTION :	*inc(int id)*
Precondition :	*/*
Postcondition :	*- if id < size2 and pres[id] = 1*
	- buf[id] = buf[id] + 1
	- buf[i] is not modified, $0 \leq i \leq$ size2 and $i \neq$ id
	- buf is not modified else -> error msg

```
void inc (char* s)
{
  int id = *((int*) s);
  if (id >= size2) fprintf(stderr,"inc : variable out of bounds\n");
  else if (pres[id]) buf[id]++;
  else fprintf(stderr,"inc : variable doesn't exist\n");
}
```

Annexe C

Adaptation de la fonction de conversion (BSM → NADF) à BSM 2.7

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <stdlib.h>
#include <bsm/audit.h>
#include <bsm/audit_record.h>
#include "gnfalib.h"

/* Original code by Aziz Mounji.
 * Heavily modified, 23 Oct 2000.
 * Last update 23 Nov 2000.
 * Ok, this is no more the original code. It was designed for Solaris 2.5, but Sun decided to
change existing names and structures in Solaris 2.7. So, we decided to rewrite this code
from scratch but we've tried to stay as close as possible to the original.We've put more
comments and more explicit name for variables but the main ideas and algorithms weren't
modified.
 */

typedef struct token_tab {
        char tok_id;
        u_short base_id;
        char *tok_addr;
} tok_ent_t;

#define TAB_LEN 62

tok_ent_t token_table[TAB_LEN] = {

 /* Control token types */

  /* 00 */ {AUT_INVALID , 1, 0} /* 0x00 */
  /* 01 */ ,{AUT_OTHER_FILE32 , 100, 0} /* 0x11 */
  /* 02 */ ,{AUT_OHEADER , 200, 0} /* 0x12 */
```

```
/* 03 */ ,{AUT_TRAILER , 300, 0} /* 0x13 */
/* 04 */ ,{AUT_HEADER32 , 400, 0} /* 0x14 */
                ,{AUT_HEADER64 , 500, 0} /* 0x74 */
// /* 05 */ ,{0 , 500, 0} /* 0xB105 */

/* Data token types */

/* 06 */ ,{AUT_DATA , 600, 0} /* 0x21 */
/* 07 */ ,{AUT_IPC , 2700, 0} /* 0x22 */
/* 08 */ ,{AUT_PATH , 2800, 0} /* 0x23 */ /* Maximum 4 path tokens in a record */
/* 09 */ ,{AUT_PATH , 2830, 0} /* 0x23 */
/* 10 */ ,{AUT_PATH , 2860, 0} /* 0x23 */
/* 11 */ ,{AUT_PATH , 2890, 0} /* 0x23 */
/* 12 */ ,{AUT_SUBJECT32 , 2900, 0} /* 0x24 */
/* 13 */ ,{AUT_SERVER32 , 3000, 0} /* 0x25 */
/* 14 */ ,{AUT_PROCESS32 , 3100, 0} /* 0x26 */
/* 15 */ ,{AUT_RETURN32 , 3200, 0} /* 0x27 */
/* 16 */ ,{AUT_TEXT , 3300, 0} /* 0x28 */ /* Maximum 4 text tokens in a record */
/* 17 */ ,{AUT_TEXT , 3330, 0} /* 0x28 */
/* 18 */ ,{AUT_TEXT , 3360, 0} /* 0x28 */
/* 19 */ ,{AUT_TEXT , 3390, 0} /* 0x28 */
/* 20 */ ,{AUT_OPAQUE , 3400, 0} /* 0x29 */
/* 21 */ ,{AUT_IN_ADDR , 3500, 0} /* 0x2A */
/* 22 */ ,{AUT_IP , 3600, 0} /* 0x2B */
/* 23 */ ,{AUT_IPORT , 3700, 0} /* 0x2C */
/* 24 */ ,{AUT_ARG32 , 3800, 0} /* 0x2D */ /* Maximum 4 arg32 tokens in a record
*/
/* 25 */ ,{AUT_ARG32 , 3830, 0} /* 0x2D */
/* 26 */ ,{AUT_ARG32 , 3860, 0} /* 0x2D */
/* 27 */ ,{AUT_ARG32 , 3890, 0} /* 0x2D */
/* 28 */ ,{AUT_SOCKET , 3900, 0} /* 0x2E */
/* 29 */ ,{AUT_SEQ , 4000, 0} /* 0x2F */

/* Modifier token types */

/* 30 */ ,{AUT_ACL , 5100, 0} /* 0x30 */
/* 31 */ ,{AUT_ATTR , 5200, 0} /* 0x31 */
/* 32 */ ,{AUT_IPC_PERM , 5300, 0} /* 0x32 */
/* 33 */ ,{AUT_LABEL , 5400, 0} /* 0x33 */
/* 34 */ ,{AUT_GROUPS , 5500, 0} /* 0x34 */
/* 35 */ ,{AUT_ILABEL , 5600, 0} /* 0x35 */
/* 36 */ ,{AUT_SLABEL , 5700, 0} /* 0x36 */
/* 37 */ ,{AUT_CLEAR , 5800, 0} /* 0x37 */
/* 38 */ ,{AUT_PRIV , 5900, 0} /* 0x38 */
//6000 is reserved for hostname
/* 39 */ ,{AUT_UPRIV , 6050, 0} /* 0x39 */
/* 40 */ ,{AUT_LIAISON , 6100, 0} /* 0x3A */
/* 41 */ ,{AUT_NEWGROUPS , 6200, 0} /* 0x3B */
/* 42 */ ,{AUT_EXEC_ARGS , 6300, 0} /* 0x3C */
/* 43 */ ,{AUT_EXEC_ENV , 6400, 0} /* 0x3D */
/* 44 */ ,{AUT_ATTR32 , 6500, 0} /* 0x2E */
```

```
  /* X windows token types */

  /* 45 */ ,{AUT_XATOM , 6600, 0} /* 0x40 */
  /* 46 */ ,{AUT_XOBJ , 6700, 0} /* 0x41 */
  /* 47 */ ,{AUT_XPROTO , 6800, 0} /* 0x42 */
  /* 48 */ ,{AUT_XSELECT , 6900, 0} /* 0x43 */

  /* Command token types */

  /* 49 */ ,{AUT_CMD , 7000, 0} /* 0x50 */
  /* 50 */ ,{AUT_EXIT , 7100, 0} /* 0x51 */

  /* Solaris64 token types */

  /* 51 */ ,{AUT_ARG64 , 8200, 0} /* 0x71 */
  /* 52 */ ,{AUT_ARG64 , 8230, 0} /* 0x71 */
  /* 53 */ ,{AUT_ARG64 , 8260, 0} /* 0x71 */
  /* 54 */ ,{AUT_ARG64 , 8290, 0} /* 0x71 */
  /* 55 */ ,{AUT_RETURN64 , 8300, 0} /* 0x72 */
  /* 56 */ ,{AUT_ATTR64 , 8400, 0} /* 0x73 */
  // /* 54 */ ,{AUT_HEADER64 , 8500, 0} /* 0x74 */
  ,{0,0,0}
  /* 58 */ ,{AUT_SUBJECT64 , 8600, 0} /* 0x75 */
  /* 59 */ ,{AUT_SERVER64 , 8700, 0} /* 0x76 */
  /* 60 */ ,{AUT_PROCESS64 , 8800, 0} /* 0x77 */
  /* 61 */ ,{AUT_OTHER_FILE64 , 8900, 0} /* 0x78 */
};



lookup_tok(char id, tok_ent_t *tab, int len)
{
        int i = 0;
        int found = 0;

        while ( !found && i < len)
                if (tab[i++].tok_id == id)
                        found = 1;
        if ( !found) {
                fprintf(stderr, "unknown token id %d", AUT_OTHER_FILE32);
                exit(1);
        }
        return i-1;
}

/* prec points to the buffer containing the BSM record to translate
 * pbuf points to the (empty) buffer which will contain the NADF record
 * pbuflen is the maximum size of this buffer
 */
nadfconv(void *prec, int *pbuflen, char **pbuf)
```

```
{

  int token_id;
  int BSM_record_ttl; // Total length of the BSM record to translate into NADF.
  char *BSM_crt_pos; // Current position into the BSM buffer.
  char *alpha = *pbuf + sizeof(int);

/* alpha is where we are going. We do not start writing NADF fields directly at the
beginning of pbuf. We leave a blank of sizeof(int) where we will put the size of the whole
NADF record. */

  int k;
  int base;

  int nb_path=0;
  int nb_text=0;
  int nb_arg32=0;
  int nb_arg64=0;

  unsigned short dummy_short;
  unsigned char dummy_char, dummy_char2;
  unsigned long dummy_long, dummy_long2, dummy_long3, dummy_long4;
  char * dummy_string;
  int dummy_int;

  token_id = * (char *) prec; //Every BSM token starts with a token_id.
  BSM_crt_pos = (char *) prec; //We are at the beginning of the BSM buffer.

  for (k=0; k<TAB_LEN; k++)
    token_table[k].tok_addr = 0;


  if (token_id == AUT_OTHER_FILE32) {
    /* This token starts and ends every BSM audit file. It's structure is (from
audit.log(4)) :
     * token ID char 1 byte 0x100
     * seconds of time uint_t 4 bytes
     * milliseconds of time uint_t 4 bytes
     * file name length short 2 bytes
     * file pathname null terminated string
     *
     * Usually, an event is logged into a record and a record contains tokens. The first
token of a record is the header token. But here, it's not the case : the file token is not part
of a record and there is no header token. The file token can be considered like a 'record' on
itself. Note that the pathname in the file token heading an audit file points to the previous
audit file and the pathname in the file token trailing an audit file points to the next audit
file.
     */

    memcpy(&dummy_short, (char *)prec+1+4+4, 2); //dummy_short now contains
the length of the pathname.
```

```
    BSM_record_ttl = 1+4+4+2+dummy_short; //BSM_record_ttl now contains the
length of the BSM record to translate.
  }
  else if ((token_id != AUT_HEADER32) && (token_id != AUT_HEADER64)) {
    /* If we are here, it's because we are not dealing with a file token. So, we must be in
presence of a record. Every record must begin with a header token. Here, it's not the case,
so it's not a valid BSM audit and it is useless to go further ...
     */
    fprintf(stderr,"[Error : conv_bsm.c] Header token not found.\n");
    fprintf(stderr," Expecting a header token ID (0x14 or 0x74) \n");
    fprintf(stderr," but found token ID %d (decimal) / 0x%x (hexa). \n",
token_id, token_id);
    fprintf(stderr,"\n");
    exit(1);
  }
  else {
    /* We are thus dealing with a header token. So, it's the beginning of a new record. But
what's the size of this record ?
     * The second field of the header token shows this size. It's structure is :
     * token ID char 1 byte
     * record byte count ulong_t 4 bytes
     * version # char 1 byte
     * event ID ushort_t 2 bytes
     * ID modifier ushort_t 2 bytes
     * seconds of time uint_t 4 bytes
     * milliseconds of time uint_t 4 bytes
     */
    memcpy(&BSM_record_ttl, (char*)prec+1, sizeof(long)); //BSM_record_ttl now
contains the length of the BSM record to translate.
  }

  /* Put some comments here */


  #ifdef DEBUG
    fprintf(stderr,"Parsing record :");
  #endif

  do {
    token_id = *BSM_crt_pos;

    #ifdef DEBUG
      fprintf(stderr, " 0x%x", token_id);
    #endif

    switch(token_id) {


      /* Control token types */
```

```
case(AUT_INVALID) :
  /* token ID | length | data
   *  1 byte     2 bytes    n bytes
   */
  k = 0;
  token_table[k].tok_addr = BSM_crt_pos;
  memcpy(&dummy_short, BSM_crt_pos+1, 2);
  BSM_record_ttl -= 1+2+dummy_short;
  BSM_crt_pos += 1+2+dummy_short;
  break;

case(AUT_OTHER_FILE32) :
  /* token ID | date & time | name length | file name
   *  1 byte      8 bytes       2 bytes      n bytes
   */
  k = 1;
  token_table[k].tok_addr = BSM_crt_pos;
  memcpy(&dummy_short, BSM_crt_pos+1+4+4, 2); //dummy_short contains
the size of the file name.
  BSM_record_ttl -= 1+4+4+2+dummy_short; //Decrease the size of the record
left to be read.
  BSM_crt_pos += 1+4+4+2+dummy_short; //... and advance into the record.
  break;

case(AUT_OHEADER) :
  k = 2;
  token_table[k].tok_addr = BSM_crt_pos;
  fprintf(stderr,"Found the undocumented AUT_OHEADER token. Not yet
implemented...\n");
  exit(1);
  break;

case(AUT_TRAILER) :
  /* token ID | pad number | byte count
   *  1 byte      2 bytes     4 bytes
   */
  k = 3;
  token_table[k].tok_addr = BSM_crt_pos;
  BSM_record_ttl -= 1+2+4;
  BSM_crt_pos += 1+2+4;
  break;

case(AUT_HEADER32) :
  /* token ID | byte count | version # | event ID | ID modifier | date and time
   *  1 byte      4 bytes     1 byte     2 bytes      2 bytes        8 bytes
   */
  k = 4;
  token_table[k].tok_addr = BSM_crt_pos;
  BSM_record_ttl -= 1+4+1+2+2+4+4;
  BSM_crt_pos += 1+4+1+2+2+4+4;
  break;
```

```
    case(AUT_TRAILER_MAGIC) :
      k = 5;
      token_table[k].tok_addr = BSM_crt_pos;
      fprintf(stderr,"Found the undocumented AUT_TRAILER_MAGIC token. Not yet
implemented...\n");
      exit(1);
      break;


      /* Data token types */


    case(AUT_DATA) :
      /* The AUT_DATA token is referenced in the SunSHIELD BSM guide under the
name arbitrary (data) token.
       *
       * token ID | print format | item size | # items | item 1 | ... | item n
       *  1 bytes      1 byte       1 byte     1 byte   n bytes        n bytes

       */
      k = 6;
      token_table[k].tok_addr = BSM_crt_pos;
      switch(*(BSM_crt_pos + 1+1)) {
      case(AUR_BYTE) : /* and AUR_CHAR : they both have the same value */
        BSM_record_ttl -= 1+1+1+1+ (* (BSM_crt_pos+1+1+1)) * 1;
        BSM_crt_pos += 1+1+1+1+ (* (BSM_crt_pos+1+1+1)) * 1;
        break;

    /*
      case(AUR_CHAR) :
        BSM_record_ttl -= 1+1+1+1+ (* (BSM_crt_pos+1+1+1)) * 1;
        BSM_crt_pos += 1+1+1+1+ (* (BSM_crt_pos+1+1+1)) * 1;
        break;
    */

      case(AUR_SHORT) :
        BSM_record_ttl -= 1+1+1+1+ (* (BSM_crt_pos+1+1+1)) * 2;
        BSM_crt_pos += 1+1+1+1+ (* (BSM_crt_pos+1+1+1)) * 2;
        break;

      case(AUR_INT) : /* and AUR_INT32 : they both have the same value */
        BSM_record_ttl -= 1+1+1+1+ (* (BSM_crt_pos+1+1+1)) * 4;
        BSM_crt_pos += 1+1+1+1+ (* (BSM_crt_pos+1+1+1)) * 4;
        break;

    /*
      case(AUR_INT32) :
        BSM_record_ttl -= 1+1+1+1+ (* (BSM_crt_pos+1+1+1)) * 4;
        BSM_crt_pos += 1+1+1+1+ (* (BSM_crt_pos+1+1+1)) * 4;
        break;
```

```
    */

      case(AUR_INT64) :
        BSM_record_ttl -= 1+1+1+1+ (* (BSM_crt_pos+1+1+1)) * 8;
        BSM_crt_pos += 1+1+1+1+ (* (BSM_crt_pos+1+1+1)) * 8;
        break;

      default :
        fprintf(stderr,"I've found an invalid data type into an Arbitrary Data
token.\n");
        fprintf(stderr,"Valid data types are AUR_BYTE (0x00), AUR_CHAR (0x00),
AUR_SHORT (0x01), \n");
        fprintf(stderr,"AUR_INT (0x02), AUR_INT32 (0x02) and AUR_INT64
(0x03).\n");
        fprintf(stderr,"Value found was %d (decimal) / %x
(hexa).\n",BSM_crt_pos+1+1, BSM_crt_pos+1+1);
      }
      break;

    case(AUT_IPC) :
      /* token ID | IPC object type | IPC object ID
       *  1 byte        1 byte          4 bytes
       */
      k = 7;
      token_table[k].tok_addr = BSM_crt_pos;
      BSM_record_ttl -= 1+1+4;
      BSM_crt_pos += 1+1+4;
      break;

    case(AUT_PATH) :
      /* token ID | path length | path
       *  1 byte      2 bytes     n bytes
       */
      if (nb_path <= 3) {
        k = 8+nb_path;
        token_table[k].tok_addr = BSM_crt_pos;
        memcpy(&dummy_short, BSM_crt_pos +1, 2);
        BSM_record_ttl -= 1+2+dummy_short;
        BSM_crt_pos += 1+2+dummy_short;
      }
      else {
        fprintf(stderr,"[Error : conv_bsm.c] Too much path tokens into this
record.\n");
        fprintf(stderr," Analysis will continue but path token number %d will be
dropped.\n",nb_path);
        memcpy(&dummy_short, BSM_crt_pos +1, 2);
        BSM_record_ttl -= 1+2+dummy_short;
        BSM_crt_pos += 1+2+dummy_short;
      }
      nb_path++;
      break;
```

```
    case(AUT_SUBJECT32) :
      /* token ID | audit ID | user ID | group ID | ru ID | rg ID | proc ID
       *  1 byte     4 bytes   4 bytes   4 bytes   4 bytes 4 bytes  4 bytes
         | sess ID | device ID | machine ID
           4 bytes    4 bytes     4 bytes
       */
      k = 12;
      token_table[k].tok_addr = BSM_crt_pos;
      BSM_record_ttl -= 1+4+4+4+4+4+4+4+4+4;
      BSM_crt_pos += 1+4+4+4+4+4+4+4+4+4;
      break;

    case(AUT_SERVER32) :
      k = 13;
      token_table[k].tok_addr = BSM_crt_pos;
      fprintf(stderr,"Found the undocumented AUT_SERVER32 token. Not yet
implemented...\n");
      exit(1);
      break;

    case(AUT_PROCESS32) :
      /* token ID | audit ID | user ID | group ID | ru ID | rg ID | proc ID
       *  1 byte     4 bytes   4 bytes   4 bytes   4 bytes 4 bytes  4 bytes
         | sess ID | device ID | machine ID
           4 bytes    4 bytes     4 bytes
       */
      k = 14;
      token_table[k].tok_addr = BSM_crt_pos;
      BSM_record_ttl -= 1+4+4+4+4+4+4+4+4+4;
      BSM_crt_pos += 1+4+4+4+4+4+4+4+4+4;
      break;

    case(AUT_RETURN32) :
      /* token ID | process error | process value
       *  1 byte       1 byte          4 byte
       */
      k = 15;
      token_table[k].tok_addr = BSM_crt_pos;
      BSM_record_ttl -= 1+1+4;
      BSM_crt_pos += 1+1+4;
      break;

    case(AUT_TEXT) :
      /* token ID | text length | text string
       *  1 byte       1 byte       n bytes
       * /=> audit.log(4) says it's 2 bytes...
       */
      if (nb_text <= 3) {
        k = 16+nb_text;
        token_table[k].tok_addr = BSM_crt_pos;
```

```
        memcpy(&dummy_short, BSM_crt_pos+1, 2);
        BSM_record_ttl -= 1+2+dummy_short;
        BSM_crt_pos += 1+2+dummy_short;
      }
      else {
        fprintf(stderr,"[Error : conv_bsm.c] Too much text tokens into this
record.\n");
        fprintf(stderr," Analysis will continue but text token number %d will be
dropped.\n",nb_text);
        fprintf(stderr," \n");
        memcpy(&dummy_short, BSM_crt_pos+1, 2);
        BSM_record_ttl -= 1+2+dummy_short;
        BSM_crt_pos += 1+2+dummy_short;
      }
      nb_text++;
      break;

    case(AUT_OPAQUE) :
      /* token ID | data length | data bytes
       *  1 byte      2 bytes       n bytes
       */
      k = 20;
      token_table[k].tok_addr = BSM_crt_pos;
      memcpy(&dummy_short, BSM_crt_pos+1, 2);
      BSM_record_ttl -= 1+2+dummy_short;
      BSM_crt_pos += 1+2+dummy_short;
      break;

    case(AUT_IN_ADDR) :
      /* token ID | Internet Address
       *  1 byte         4 bytes
       */
      k = 21;
      token_table[k].tok_addr = BSM_crt_pos;
      BSM_record_ttl -= 1+4;
      BSM_crt_pos += 1+4;
      break;

    case(AUT_IP) :
      /* token ID | IP Header
       *  1 byte    20 bytes
       */
      k = 22;
      token_table[k].tok_addr = BSM_crt_pos;
      BSM_record_ttl -= 1+20;
      BSM_crt_pos += 1+20;
      break;

    case(AUT_IPORT) :
      /* token ID | port ID
       *  1 byte    2 bytes
```

```
        */
       k = 23;
       token_table[k].tok_addr = BSM_crt_pos;
       BSM_record_ttl -= 1+2;
       BSM_crt_pos += 1+2;
       break;

     case(AUT_ARG32) :
       /* token ID | argument # | argument value | text length | text
        *  1 byte      1 byte         4 bytes        2 bytes     n bytes
        */
       if (nb_arg32 <= 3) {
         k = 24+nb_arg32;
         token_table[k].tok_addr = BSM_crt_pos;
         memcpy(&dummy_short, BSM_crt_pos+1+1+4, 2);
         BSM_record_ttl -= 1+1+4+2+dummy_short;
         BSM_crt_pos += 1+1+4+2+dummy_short;
       }
       else {
         fprintf(stderr,"[Error : conv_bsm.c] Too much arg32 tokens into this
record.\n");
         fprintf(stderr," Analysis will continue but arg32 token number %d will
be dropped.\n",nb_arg32);
         fprintf(stderr," \n");
         memcpy(&dummy_short, BSM_crt_pos+1+1+4, 2);
         BSM_record_ttl -= 1+1+4+2+dummy_short;
         BSM_crt_pos += 1+1+4+2+dummy_short;
       }
       nb_arg32++;
       break;

     case(AUT_SOCKET) :
       /* token ID | socket type | local port | local Inet address
        *  1 bytes     2 bytes       2 bytes          4 bytes
           | remote port | remote Inet address
               2 bytes           4 bytes
        */
       k = 28;
       token_table[k].tok_addr = BSM_crt_pos;
       BSM_record_ttl -= 1+2+2+4;//+2+4;
       BSM_crt_pos += 1+2+2+4;//+2+4;
       break;

     case(AUT_SEQ) :
       /* token ID | sequence number
        *  1 byte        4 bytes
        */
       k = 29;
       token_table[k].tok_addr = BSM_crt_pos;
       BSM_record_ttl -= 1+4;
       BSM_crt_pos += 1+4;
```

```
      break;


      /* Modifier token types */


    case(AUT_ACL) :
      /* token ID | ACL type | ACL ID | ACL permissions
       *  1 byte     4 bytes   4 bytes      4 bytes
       */
      k = 30;
      token_table[k].tok_addr = BSM_crt_pos;
      BSM_record_ttl -= 1+4+4+4;
      BSM_crt_pos += 1+4+4+4;
      break;

    case(AUT_ATTR) :
      /* token ID | file mode | owner UID | owner GID | file system ID
       *  1 byte     4 bytes     4 bytes     4 bytes       4 bytes
          | file inode ID | device ID
              4 bytes       4 bytes
       */
      k = 31;
      token_table[k].tok_addr = BSM_crt_pos;
      BSM_record_ttl -= 1+4+4+4+4+4+4;
      BSM_crt_pos += 1+4+4+4+4+4+4;
      break;

    case(AUT_IPC_PERM) :
      /* token ID | owner UID | owner GID | creator UID | creator GID | ipc mode
       *  1 bytes    4 bytes     4 bytes     4 bytes       4 bytes       4 bytes
          | sequence ID | IPC key
              4 bytes     4 bytes
       */
      k = 32;
      token_table[k].tok_addr = BSM_crt_pos;
      BSM_record_ttl -= 1+4+4+4+4+4+4+4;
      BSM_crt_pos += 1+4+4+4+4+4+4+4;
      break;

    case(AUT_LABEL) :
      k = 33;
      token_table[k].tok_addr = BSM_crt_pos;
      fprintf(stderr,"Found the undocumented AUT_LABEL token. Not yet
implemented...\n");
      exit(1);
      break;

    case(AUT_GROUPS) :
      /* token ID | count | group1 | ... | group n
       *  1 byte   2 bytes 4 bytes   ...  4 bytes
```

```
        * According to SunShield BSM Guide, there is no count field...
        */
       k = 34;
       token_table[k].tok_addr = BSM_crt_pos;
       memcpy(&dummy_short, BSM_crt_pos+1, 2);
       BSM_record_ttl -= 1+2+ (4*dummy_short);
       BSM_crt_pos += 1+2+ (4*dummy_short);
       break;

    case(AUT_ILABEL) :
       k = 35;
       token_table[k].tok_addr = BSM_crt_pos;
       fprintf(stderr,"Found the undocumented AUT_ILABEL token. Not yet
implemented...\n");
       exit(1);
       break;

    case(AUT_SLABEL) :
       k = 36;
       token_table[k].tok_addr = BSM_crt_pos;
       fprintf(stderr,"Found the undocumented AUT_SLABEL token. Not yet
implemented...\n");
       exit(1);
       break;

    case(AUT_CLEAR) :
       k = 37;
       token_table[k].tok_addr = BSM_crt_pos;
       fprintf(stderr,"Found the undocumented AUT_CLEAR token. Not yet
implemented...\n");
       exit(1);
       break;

    case(AUT_PRIV) :
       k = 38;
       token_table[k].tok_addr = BSM_crt_pos;
       fprintf(stderr,"Found the undocumented AUT_PRIV token. Not yet
implemented...\n");
       exit(1);
       break;

    case(AUT_UPRIV) :
       k = 39;
       token_table[k].tok_addr = BSM_crt_pos;
       fprintf(stderr,"Found the undocumented AUT_UPRIV token. Not yet
implemented...\n");
       exit(1);
       break;

    case(AUT_LIAISON) :
       k = 40;
```

```
        token_table[k].tok_addr = BSM_crt_pos;
        fprintf(stderr,"Found the undocumented AUT_LIAISON token. Not yet
implemented...\n");
        exit(1);
        break;

    case(AUT_NEWGROUPS) :
        /* token ID | count | group1 | ... | group n
         *  1 byte   2 bytes 4 bytes   ...  4 bytes
         */
        k = 41;
        token_table[k].tok_addr = BSM_crt_pos;
        memcpy(&dummy_short, BSM_crt_pos+1, 2);
        BSM_record_ttl -= 1+2+ (4*dummy_short);
        BSM_crt_pos += 1+2+ (4*dummy_short);
        break;

    case(AUT_EXEC_ARGS) :
        /* token ID | count | env_args
         *  1 byte   4 bytes  count * n bytes
         * count is not the size of the arguments passed to the exec call but is the
         * number of arguments !
         */
        k = 42;
        token_table[k].tok_addr = BSM_crt_pos;
        memcpy(&dummy_long, BSM_crt_pos+1, 4);
        // dummy_long now contains the number of strings presents in the BSM record

        // dummy_long2 counts the strings encountered
        // dummy_string points to the first string

        // When we have found all the strings (that is : when dummy_long2 >=
dummy_long) we exit from the loop.

        for(dummy_long2 = 0, dummy_string = BSM_crt_pos+1+4; dummy_long2 <
dummy_long; dummy_long2++) {
            dummy_string += strlen(dummy_string)+1; //dummy_string points to the
next string.
        }

        // Now dummy_string points to the first byte after the last string.
        BSM_record_ttl -= 1+4+ (dummy_string - (BSM_crt_pos +1+4));
        BSM_crt_pos += 1+4+ (dummy_string - (BSM_crt_pos +1+4));
        break;

    case(AUT_EXEC_ENV) :
        /* token ID | count | env_args
         *  1 byte   4 bytes  count * n bytes
         * count is not the size of the arguments passed to the exec call but is the
         * number of arguments !
         */
```

```
        k = 43;
        token_table[k].tok_addr = BSM_crt_pos;
        memcpy(&dummy_long, BSM_crt_pos+1, 4);
        // dummy_long now contains the number of strings presents in the BSM record

        // dummy_long2 counts the strings encountered
        // dummy_string points to the first string

        // When we have found all the strings (that is : when dummy_long2 >=
dummy_long) we exit from the loop.

        for(dummy_long2 = 0, dummy_string = BSM_crt_pos+1+4; dummy_long2 <
dummy_long; dummy_long2++) {
          dummy_string += strlen(dummy_string)+1; //dummy_string points to the
next string.
        }

        // Now dummy_string points to the first byte after the last string.
        BSM_record_ttl -= 1+4+ (dummy_string - (BSM_crt_pos +1+4));
        BSM_crt_pos += 1+4+ (dummy_string - (BSM_crt_pos +1+4));
        break;

      case(AUT_ATTR32) :
        /* token ID | file mode | owner UID | owner GID | file system ID
         *  1 byte     4 bytes     4 bytes     4 bytes       4 bytes
           | file inode ID | device ID
                8 bytes       4 bytes
          Seems to be (another) bug in BSM doc. 2 tokens are defines : AUT_ATTR and
AUT_ATTR32, they do NOT have the same structure. AUT_ATTR32 seems to have
additional 4 bytes. I think that the node ID is coded on 8 bytes instead of 4 bytes (at least,
that's what it's said in the 2.8 version of audit.log(4).
         */
        k = 44;
        token_table[k].tok_addr = BSM_crt_pos;
        BSM_record_ttl -= 1+4+4+4+4+8+4;
        BSM_crt_pos += 1+4+4+4+4+8+4;
        break;


        /* X windows token types */


      case(AUT_XATOM) :
        k = 45;
        token_table[k].tok_addr = BSM_crt_pos;
        fprintf(stderr,"Found the undocumented AUT_XATOM token. Not yet
implemented...\n");
        exit(1);
        break;

      case(AUT_XOBJ) :
```

```
      k = 46;
      token_table[k].tok_addr = BSM_crt_pos;
      fprintf(stderr,"Found the undocumented AUT_XOBJ token. Not yet
implemented...\n");
      exit(1);
      break;

    case(AUT_XPROTO) :
      k = 47;
      token_table[k].tok_addr = BSM_crt_pos;
      fprintf(stderr,"Found the undocumented AUT_XPROTO token. Not yet
implemented...\n");
      exit(1);
      break;

    case(AUT_XSELECT) :
      k = 48;
      token_table[k].tok_addr = BSM_crt_pos;
      fprintf(stderr,"Found the undocumented AUT_XSELECT token. Not yet
implemented...\n");
      exit(1);
      break;


      /* Command token types */


    case(AUT_CMD) :
      k = 49;
      token_table[k].tok_addr = BSM_crt_pos;
      fprintf(stderr,"Found the undocumented AUT_CMD token. Not yet
implemented...\n");
      exit(1);
      break;

    case(AUT_EXIT) :
      /* token ID | status | return value
       *  1 byte    4 bytes     4 bytes
       */
      k = 50;
      token_table[k].tok_addr = BSM_crt_pos;
      BSM_record_ttl -= 1+4+4;
      BSM_crt_pos += 1+4+4;
      break;


      /* Solaris64 token types */

    case(AUT_ARG64) :
      /* token ID | argument # | argument value | text length | text
       *  1 byte      1 byte         8 bytes        2 bytes    n bytes
```

```
        */
       if (nb_arg64 <= 3) {
         k = 51+nb_arg64;
         token_table[k].tok_addr = BSM_crt_pos;
         memcpy(&dummy_short, BSM_crt_pos+1+1+8, 2);
         BSM_record_ttl -= 1+1+8+2+dummy_short;
         BSM_crt_pos += 1+1+8+2+dummy_short;
       }
       else {
         fprintf(stderr,"[Error : conv_bsm.c] Too much arg64 tokens into this
record.\n");
         fprintf(stderr," Analysis will continue but arg64 token number %d will
be dropped.\n",nb_arg32);
         fprintf(stderr," \n");
         memcpy(&dummy_short, BSM_crt_pos+1+1+8, 2);
         BSM_record_ttl -= 1+1+8+2+dummy_short;
         BSM_crt_pos += 1+1+8+2+dummy_short;
       }
       nb_arg64++;
       break;

     case(AUT_RETURN64) :
       /* token ID | process error | process value
        *  1 byte       1 byte          8 byte
        */
       k = 55;
       token_table[k].tok_addr = BSM_crt_pos;
       BSM_record_ttl -= 1+1+8;
       BSM_crt_pos += 1+1+8;
       break;

     case(AUT_ATTR64) :
       /* token ID | file mode | owner UID | owner GID | file system ID
        *  1 byte     4 bytes     4 bytes     4 bytes       4 bytes
           | file inode ID | device ID
                8 bytes       8 bytes
        */
       k = 56;
       token_table[k].tok_addr = BSM_crt_pos;
       BSM_record_ttl -= 1+4+4+4+4+8+8;
       BSM_crt_pos += 1+4+4+4+4+8+8;
       break;

     case(AUT_HEADER64) :
       /* token ID | byte count | version # | event ID | ID modifier
        *  1 byte      4 bytes     1 byte     2 bytes      2 bytes
           | seconds of time | milisec of time
                8 bytes            8 bytes
        */
       k = 5;
       token_table[k].tok_addr = BSM_crt_pos;
```

```
      BSM_record_ttl -= 1+4+1+2+2+8+8;
      BSM_crt_pos += 1+4+1+2+2+8+8;
      break;

    case(AUT_SUBJECT64) :
      /* token ID | audit ID | user ID | group ID | ru ID | rg ID | proc ID
       *  1 byte     4 bytes   4 bytes   4 bytes   4 bytes 4 bytes  4 bytes
          | sess ID | device ID | machine ID
            4 bytes    8 bytes     4 bytes
       */
      k = 58;
      token_table[k].tok_addr = BSM_crt_pos;
      BSM_record_ttl -= 1+4+4+4+4+4+4+4+8+4;
      BSM_crt_pos += 1+4+4+4+4+4+4+4+8+4;
      break;

    case(AUT_SERVER64) :
      k = 59;
      token_table[k].tok_addr = BSM_crt_pos;
      fprintf(stderr,"Found the undocumented AUT_SERVER64 token. Not yet
implemented...\n");
      exit(1);
      break;

    case(AUT_PROCESS64) :
      /* token ID | audit ID | user ID | group ID | ru ID | rg ID | proc ID
       *  1 byte     4 bytes   4 bytes   4 bytes   4 bytes 4 bytes  4 bytes
          | sess ID | device ID | machine ID
            4 bytes    8 bytes     4 bytes
       */
      k = 60;
      token_table[k].tok_addr = BSM_crt_pos;
      BSM_record_ttl -= 1+4+4+4+4+4+4+4+8+4;
      BSM_crt_pos += 1+4+4+4+4+4+4+4+8+4;
      break;

    case(AUT_OTHER_FILE64) :
      k = 61;
      token_table[k].tok_addr = BSM_crt_pos;
      fprintf(stderr,"Found the undocumented AUT_OTHER_FILE64 token. Not yet
implemented...\n");
      exit(1);
      break;

    default :
      fprintf(stderr,"Found an unknown token : %d (decimal) / 0x%x
(hexa).\n",token_id);
      exit(1);
    }
  } while(BSM_record_ttl > 0);
```

```
#ifdef DEBUG
    fprintf(stderr, ".\n");
#endif

    /* NADF making comments */
  /* The token table is now filled. We will scan it in order to create the NADF record.
   *
   * - If the field tok_addr is NULL for an entry in the table, it means that this token was
not present into the BSM record. We skip it.
   * - Otherwise, the field tok_addr points to the beginning of the token (which is located
'into' prec, the BSM buffer). We will convert this token into NADF fields.
   * Each BSM token contains several fields. Each BSM field will be converted into an
NADF field. Basically, we simply create an header and we append it the exact copy of the
value of the original field. This header contains an ID and the size of the field. The ID is
not the original Token ID used in BSM. This is a brand new ID (this is the ID referenced
in the DDF file). The new NADF ID to use for the first BSM field of the token is
contained in base_id. And for the next fields of the record, we just keep incrementing this
base. It sounds complicated but it's quite simple. Take a look on the code to fix your ideas...
   */

  for (k=0; k<TAB_LEN; k++) {
    if( !token_table[k].tok_addr)
      continue;

    token_id = *(token_table[k].tok_addr);
    prec = (void *) token_table[k].tok_addr;

    switch(token_id) {


      /* Control token types */


    case(AUT_INVALID) :
      exit(1);
      break;

    case(AUT_OTHER_FILE32) :
      /* token ID | date & time | name length | file name
       *  1 byte      8 bytes       2 bytes      n bytes
       */
      base = token_table[k].base_id;
      put_bytes(&alpha, base++, 1, (char *) prec, pbuflen, pbuf);
      put_bytes(&alpha, base++, 4, (char *) prec+1, pbuflen, pbuf);
      put_bytes(&alpha, base++, 4, (char *) prec+1+4, pbuflen, pbuf);
      put_bytes(&alpha, base++, 2, (char *) prec+1+4+4, pbuflen, pbuf);
      memcpy(&dummy_short, (char *) prec+1+4+4, 2);
      put_bytes(&alpha, base++, dummy_short, prec+1+4+4+2, pbuflen,pbuf);
      break;

    case(AUT_OHEADER) :
```

```
        exit(1);
        break;

    case(AUT_TRAILER) :
        /* token ID | pad number | byte count
         *  1 byte      2 bytes     4 bytes
         */
        base = token_table[k].base_id;
        put_bytes(&alpha, base++, 1, (char *) prec, pbuflen, pbuf);
        put_bytes(&alpha, base++, 2, (char *) prec+1, pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1+2, pbuflen, pbuf);
        break;

    case(AUT_HEADER32) :
        /* token ID | byte count | version # | event ID | ID modifier | date and time
         *  1 byte      4 bytes     1 byte     2 bytes      2 bytes        8 bytes
         */
        base = token_table[k].base_id;
        put_bytes(&alpha, base++, 1, (char *) prec, pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1, pbuflen, pbuf);
        put_bytes(&alpha, base++, 1, (char *) prec+1+4, pbuflen, pbuf);
        put_bytes(&alpha, base++, 2, (char *) prec+1+4+1, pbuflen, pbuf);
        put_bytes(&alpha, base++, 2, (char *) prec+1+4+1+2, pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1+4+1+2+2, pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1+4+1+2+2+4, pbuflen, pbuf);
        break;

    case(AUT_TRAILER_MAGIC) :
        exit(1);
        break;


        /* Data token types */


    case(AUT_DATA) :
        /* The AUT_DATA token is referenced in the SunSHIELD BSM guide under the
name arbitrary (data) token.
         *
         * token ID | print format | item size | # items | item 1 | ... | item n
         *  1 bytes      1 byte       1 byte     1 byte   n bytes        n bytes

         */
        base = token_table[k].base_id;
        put_bytes(&alpha, base++, 1, (char *) prec, pbuflen, pbuf);
        put_bytes(&alpha, base++, 1, (char *) prec+1, pbuflen, pbuf);
        put_bytes(&alpha, base++, 1, (char *) prec+1+1, pbuflen, pbuf);
        put_bytes(&alpha, base++, 1, (char *) prec+1+1+1, pbuflen, pbuf);

        switch(*(char *) prec+1+1) {
        case(AUR_BYTE) : /* and AUR_CHAR : they both have the same value */
```

```
        memcpy(&dummy_short, (char *) prec+1+1+1, 1);
        for(dummy_int = 0; dummy_int < dummy_short; dummy_int++) {
          put_bytes(&alpha, base++, 1, (char *) prec+1+1+1+1 + dummy_short,
pbuflen, pbuf);
        }
        break;

      case(AUR_SHORT) :
        base += 256;
        memcpy(&dummy_short, (char *) prec+1+1+1, 1);
        for(dummy_int = 0; dummy_int < dummy_short; dummy_int++) {
          put_bytes(&alpha, base++, 2, (char *) prec+1+1+1+1 + 2*dummy_short,
pbuflen, pbuf);
        }
        break;

      case(AUR_INT) : /* and AUR_INT32 : they both have the same value */
        base += 512;
        memcpy(&dummy_short, (char *) prec+1+1+1, 1);
        for(dummy_int = 0; dummy_int < dummy_short; dummy_int++) {
          put_bytes(&alpha, base++, 4, (char *) prec+1+1+1+1 + 4*dummy_short,
pbuflen, pbuf);
        }
        break;

      case(AUR_INT64) :
        base += 768;
        memcpy(&dummy_short, (char *) prec+1+1+1, 1);
        for(dummy_int = 0; dummy_int < dummy_short; dummy_int++) {
          put_bytes(&alpha, base++, 4, (char *) prec+1+1+1+1 + 4*dummy_short,
pbuflen, pbuf);
        }
        break;
      }
      break;

    case(AUT_IPC) :
      /* token ID | IPC object type | IPC object ID
       *  1 byte        1 byte          4 bytes
       */
      base = token_table[k].base_id;
      put_bytes(&alpha, base++, 1, (char *) prec, pbuflen, pbuf);
      put_bytes(&alpha, base++, 1, (char *) prec+1, pbuflen, pbuf);
      put_bytes(&alpha, base++, 4, (char *) prec+1+1, pbuflen, pbuf);
      break;

    case(AUT_PATH) :
      /* token ID | path length | path
       *  1 byte      2 bytes     n bytes
       */
      base = token_table[k].base_id;
```

```
        put_bytes(&alpha, base++, 1, (char *) prec, pbuflen, pbuf);
        memcpy(&dummy_short, (char *) prec+1, 2); //these 2 lines were inverted!
        put_bytes(&alpha, base++, 2, (char *) &dummy_short, pbuflen, pbuf);
        put_bytes(&alpha, base++, dummy_short, (char *) prec+1+2, pbuflen, pbuf);
        break;

    case(AUT_SUBJECT32) :
        /* token ID | audit ID | user ID | group ID | ru ID | rg ID | proc ID
         *  1 byte     4 bytes   4 bytes   4 bytes   4 bytes 4 bytes  4 bytes
           | sess ID | device ID | machine ID
             4 bytes    4 bytes     4 bytes
         */
        base = token_table[k].base_id;
        put_bytes(&alpha, base++, 1, (char *) prec, pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1, pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1+4 , pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1+4+4 , pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1+4+4+4 , pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1+4+4+4+4 , pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1+4+4+4+4+4 , pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1+4+4+4+4+4+4 , pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1+4+4+4+4+4+4+4 , pbuflen,
pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1+4+4+4+4+4+4+4+4 , pbuflen,
pbuf);
        break;

    case(AUT_SERVER32) :
        exit(1);
        break;

    case(AUT_PROCESS32) :
        /* token ID | audit ID | user ID | group ID | ru ID | rg ID | proc ID
         *  1 byte     4 bytes   4 bytes   4 bytes   4 bytes 4 bytes  4 bytes
           | sess ID | device ID | machine ID
             4 bytes    4 bytes     4 bytes
         */
        base = token_table[k].base_id;
        put_bytes(&alpha, base++, 1, (char *) prec, pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1, pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1+4 , pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1+4+4 , pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1+4+4+4 , pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1+4+4+4+4 , pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1+4+4+4+4+4 , pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1+4+4+4+4+4+4 , pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1+4+4+4+4+4+4+4 , pbuflen,
pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1+4+4+4+4+4+4+4+4 , pbuflen,
pbuf);
        break;
```

```
case(AUT_RETURN32) :
  /* token ID | process error | process value
   *  1 byte       1 byte          4 byte
   */
  base = token_table[k].base_id;
  put_bytes(&alpha, base++, 1, (char *) prec, pbuflen, pbuf);
  put_bytes(&alpha, base++, 1, (char *) prec+1, pbuflen, pbuf);
  put_bytes(&alpha, base++, 4, (char *) prec+1+1, pbuflen, pbuf);
  break;

case(AUT_TEXT) :
  /* token ID | text length | text string
   *  1 byte       1 byte       n bytes
   */
  base = token_table[k].base_id;
  put_bytes(&alpha, base++, 1, (char *) prec, pbuflen, pbuf);
  put_bytes(&alpha, base++, 2, (char *) prec+1, pbuflen, pbuf);
  memcpy(&dummy_short, (char *) prec+1, 2);
  put_bytes(&alpha, base++, dummy_short, (char *) prec+1+1, pbuflen, pbuf);
  break;

case(AUT_OPAQUE) :
  /* token ID | data length | data bytes
   *  1 byte      2 bytes       n bytes
   */
  base = token_table[k].base_id;
  put_bytes(&alpha, base++, 1, (char *) prec, pbuflen, pbuf);
  put_bytes(&alpha, base++, 2, (char *) prec+1, pbuflen, pbuf);
  memcpy(&dummy_short, (char *) prec+1, 2);
  put_bytes(&alpha, base++, dummy_short, (char *) prec+1+2, pbuflen, pbuf);
  break;

case(AUT_IN_ADDR) :
  /* token ID | Internet Address
   *  1 byte         4 bytes
   */
  base = token_table[k].base_id;
  put_bytes(&alpha, base++, 1, (char *) prec, pbuflen, pbuf);
  put_bytes(&alpha, base++, 4, (char *) prec+1, pbuflen, pbuf);
  break;

case(AUT_IP) :
  /* token ID | IP Header
   *  1 byte    20 bytes
   */
  base = token_table[k].base_id;
  put_bytes(&alpha, base++, 1, (char *) prec, pbuflen, pbuf);

  memcpy(&dummy_char, (char *) prec+1,1);
  dummy_char2 = (dummy_char << 4 ) >> 4; // Header Length
```

```
        dummy_char = dummy_char ≫ 4; // Version
        put_bytes(&alpha, base++, 1, &dummy_char, pbuflen, pbuf);
        put_bytes(&alpha, base++, 1, &dummy_char2, pbuflen, pbuf);

        put_bytes(&alpha, base++, 1, (char *) prec+1+1, pbuflen, pbuf); // Type of
service
        put_bytes(&alpha, base++, 2, (char *) prec+1+1+1, pbuflen, pbuf); // Total
length
        put_bytes(&alpha, base++, 2, (char *) prec+1+1+1+2, pbuflen, pbuf); //
Identification

        memcpy(&dummy_char, (char *) prec+1+1+1+2+2, 1);
        dummy_char = dummy_char ≫ 5;
        put_bytes(&alpha, base++, 1, &dummy_char, pbuflen, pbuf); // Flags

        memcpy(&dummy_short, (char *) prec+1+1+1+2+2, 2);
        dummy_short = (dummy_short ≪ 3) ≫ 3;
        put_bytes(&alpha, base++, 2, (char *) &dummy_short, pbuflen, pbuf); //
Fragment Offset

        put_bytes(&alpha, base++, 1, (char *) prec+1+1+1+2+2, pbuflen, pbuf); //
Time to live
        put_bytes(&alpha, base++, 1, (char *) prec+1+1+1+2+2+1, pbuflen, pbuf); //
Protocol
        put_bytes(&alpha, base++, 2, (char *) prec+1+1+1+2+2+1+1, pbuflen, pbuf);
// Header checksum
        put_bytes(&alpha, base++, 4, (char *) prec+1+1+1+2+2+1+1+2, pbuflen,
pbuf); // Source address
        put_bytes(&alpha, base++, 4, (char *) prec+1+1+1+2+2+1+1+2+4, pbuflen,
pbuf); // Destination address




    put_bytes(&alpha, base++, 20, (char *) prec+1, pbuflen, pbuf);
      break;

    case(AUT_IPORT) :
      /* token ID | port ID
       *  1 byte    2 bytes
       */
      base = token_table[k].base_id;
      put_bytes(&alpha, base++, 1, (char *) prec, pbuflen, pbuf);
      put_bytes(&alpha, base++, 2, (char *) prec+1, pbuflen, pbuf);
      break;

    case(AUT_ARG32) :
      /* token ID | argument # | argument value | text length | text
       *  1 byte      1 byte         4 bytes        2 bytes    n bytes
       */
      base = token_table[k].base_id;
```

```
        put_bytes(&alpha, base++, 1, (char *) prec, pbuflen, pbuf);
        put_bytes(&alpha, base++, 1, (char *) prec+1, pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1+1, pbuflen, pbuf);
        put_bytes(&alpha, base++, 2, (char *) prec+1+1+4, pbuflen, pbuf);
        memcpy(&dummy_short, (char *)prec+1+1+4, 2);
        put_bytes(&alpha, base++, dummy_short, (char *) prec+1+1+4+2, pbuflen,
pbuf);
        break;

    case(AUT_SOCKET) :
        /* token ID | socket type | local port | local Inet address
         *  1 bytes     2 bytes       2 bytes          4 bytes
            | remote port | remote Inet address
                2 bytes           4 bytes
         */
        base = token_table[k].base_id;
        put_bytes(&alpha, base++, 1, (char *) prec, pbuflen, pbuf);
        put_bytes(&alpha, base++, 2, (char *) prec+1, pbuflen, pbuf);
        put_bytes(&alpha, base++, 2, (char *) prec+1+2, pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1+2+2, pbuflen, pbuf);
        put_bytes(&alpha, base++, 2, (char *) prec+1+2+2+4, pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1+2+2+4+2, pbuflen, pbuf);
        break;

    case(AUT_SEQ) :
        /* token ID | sequence number
         *  1 byte        4 bytes
         */
        base = token_table[k].base_id;
        put_bytes(&alpha, base++, 1, (char *) prec, pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1, pbuflen, pbuf);
        break;


        /* Modifier token types */


    case(AUT_ACL) :
        /* token ID | ACL type | ACL ID | ACL permissions
         *  1 byte     4 bytes   4 bytes      4 bytes
         */
        base = token_table[k].base_id;
        put_bytes(&alpha, base++, 1, (char *) prec, pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1, pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1+4, pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1+4+4, pbuflen, pbuf);
        break;

    case(AUT_ATTR) :
        /* token ID | file mode | owner UID | owner GID | file system ID
         *  1 byte     4 bytes     4 bytes     4 bytes       4 bytes
```

```
        | file inode ID | device ID
             4 bytes        4 bytes
         */
        base = token_table[k].base_id;
        put_bytes(&alpha, base++, 1, (char *) prec, pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1, pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1+4, pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1+4+4, pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1+4+4+4, pbuflen, pbuf);
        put_bytes(&alpha, base++, 8, (char *) prec+1+4+4+4+4, pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1+4+4+4+4+8, pbuflen, pbuf);
        break;

      case(AUT_IPC_PERM) :
        /* token ID | owner UID | owner GID | creator UID | creator GID | ipc mode
         *  1 bytes     4 bytes     4 bytes     4 bytes       4 bytes      4 bytes
            | sequence ID | IPC key
                4 bytes     4 bytes
         */
        base = token_table[k].base_id;
        put_bytes(&alpha, base++, 1, (char *) prec, pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1, pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1+4, pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1+4+4, pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1+4+4+4, pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1+4+4+4+4, pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1+4+4+4+4+4, pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1+4+4+4+4+4+4, pbuflen, pbuf);
        break;

      case(AUT_LABEL) :
        exit(1);
        break;

      case(AUT_GROUPS) :
        /* token ID | count | group1 | ... | group n
         *  1 byte   2 bytes 4 bytes    ...  4 bytes
         * According to SunShield BSM Guide, there is no count field...
         */
        base = token_table[k].base_id;
        put_bytes(&alpha, base++, 1, (char *) prec, pbuflen, pbuf);
        put_bytes(&alpha, base++, 2, (char *) prec+1, pbuflen, pbuf);
        memcpy(&dummy_short, (char *) prec+1, 2);
        for(dummy_int = 0; dummy_int < dummy_short; dummy_int++) {
          put_bytes(&alpha, base++, 4, (char *) prec+1+2 + 4*dummy_short, pbuflen,
pbuf);
        }
        break;

      case(AUT_ILABEL) :
        exit(1);
```

```
        break;

      case(AUT_SLABEL) :
        exit(1);
        break;

      case(AUT_CLEAR) :
        exit(1);
        break;

      case(AUT_PRIV) :
        exit(1);
        break;

      case(AUT_UPRIV) :
        exit(1);
        break;

      case(AUT_LIAISON) :
        exit(1);
        break;

      case(AUT_NEWGROUPS) :
        /* token ID | count | group1 | ... | group n
         *  1 byte   2 bytes 4 bytes   ...  4 bytes
         */
        base = token_table[k].base_id;
        put_bytes(&alpha, base++, 1, (char *) prec, pbuflen, pbuf);
        put_bytes(&alpha, base++, 1, (char *) prec, pbuflen, pbuf);
        put_bytes(&alpha, base++, 2, (char *) prec+1, pbuflen, pbuf);
        memcpy(&dummy_short, (char *) prec+1, 2);
        for(dummy_int = 0; dummy_int < dummy_short; dummy_int++) {
          put_bytes(&alpha, base++, 4, (char *) prec+1+2 + 4*dummy_short, pbuflen,
pbuf);
        }
        break;

      case(AUT_EXEC_ARGS) :
        /* token ID | count | env_args
         *  1 byte   4 bytes  count * n bytes
         * count is not the size of the arguments passed to the exec call but is the
         * number of arguments!
         */
        base = token_table[k].base_id;
        put_bytes(&alpha, base++, 1, (char *) prec, pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1, pbuflen, pbuf);

        memcpy(&dummy_long,(char *) prec+1, 4); //dummy_long contains the number
of args

        dummy_long3 = 0; //Total size of the args string
```

```
        for(dummy_long2 = 0, dummy_string = (char *) prec+1+4; dummy_long2 <
dummy_long; dummy_long2++) {
            dummy_long4 = strlen(dummy_string); //dummy_long4 is the size of the
string without the trailing \0
            //*(dummy_string + dummy_long4) = ' '; // \0 is replaced by ' ' NO MORE
IMPLEMENTED
            dummy_string = dummy_string + dummy_long4 + 1; // DS2 points to the
next string
            dummy_long3 += dummy_long4+1;
        }

        put_bytes(&alpha, base++, dummy_long3, (char *) prec+1+4, pbuflen, pbuf);
        break;


    case(AUT_EXEC_ENV) :
        /* token ID | count | env_args
         *  1 byte   4 bytes  count * n bytes
         * count is not the size of the arguments passed to the exec call but is the
         * number of arguments !
         */
        base = token_table[k].base_id;
        put_bytes(&alpha, base++, 1, (char *) prec, pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1, pbuflen, pbuf);

        memcpy(&dummy_long,(char *) prec+1, 4); //dummy_long contains the number
of args

        dummy_long3 = 0; //Total size of the args string

        for(dummy_long2 = 0, dummy_string = (char *) prec+1+4; dummy_long2 <
dummy_long; dummy_long2++) {
            dummy_long4 = strlen(dummy_string); //dummy_long4 is the size of the
string without the trailing \0
            //*(dummy_string + dummy_long4) = ' '; // \0 is replaced by ' ' NO MORE
IMPLEMENTED
            dummy_string = dummy_string + dummy_long4 + 1; // DS2 points to the
next string
            dummy_long3 += dummy_long4+1;
        }

        put_bytes(&alpha, base++, dummy_long3, (char *) prec+1+4, pbuflen, pbuf);
        break;


    case(AUT_ATTR32) :
        /* token ID | file mode | owner UID | owner GID | file system ID
         *  1 byte     4 bytes     4 bytes     4 bytes       4 bytes
            | file inode ID | device ID
                8 bytes       4 bytes
```

```
    */
    base = token_table[k].base_id;
    put_bytes(&alpha, base++, 1, (char *) prec, pbuflen, pbuf);
    put_bytes(&alpha, base++, 4, (char *) prec+1, pbuflen, pbuf);
    put_bytes(&alpha, base++, 4, (char *) prec+1+4, pbuflen, pbuf);
    put_bytes(&alpha, base++, 4, (char *) prec+1+4+4, pbuflen, pbuf);
    put_bytes(&alpha, base++, 4, (char *) prec+1+4+4+4, pbuflen, pbuf);
    put_bytes(&alpha, base++, 8, (char *) prec+1+4+4+4+4, pbuflen, pbuf);
    put_bytes(&alpha, base++, 4, (char *) prec+1+4+4+4+4+8, pbuflen, pbuf);
    break;


    /* X windows token types */


  case(AUT_XATOM) :
    exit(1);
    break;

  case(AUT_XOBJ) :
    exit(1);
    break;

  case(AUT_XPROTO) :
    exit(1);
    break;

  case(AUT_XSELECT) :
    exit(1);
    break;


    /* Command token types */


  case(AUT_CMD) :
    exit(1);
    break;

  case(AUT_EXIT) :
    /* token ID | status | return value
     *  1 byte    4 bytes     4 bytes
     */
    base = token_table[k].base_id;
    put_bytes(&alpha, base++, 1, (char *) prec, pbuflen, pbuf);
    put_bytes(&alpha, base++, 4, (char *) prec+1, pbuflen, pbuf);
    put_bytes(&alpha, base++, 4, (char *) prec+1+4, pbuflen, pbuf);
    break;


    /* Solaris64 token types */
```

```
    case(AUT_ARG64) :
      /* token ID | argument # | argument value | text length | text
       *  1 byte      1 byte         8 bytes        2 bytes    n bytes
       */
      base = token_table[k].base_id;
      put_bytes(&alpha, base++, 1, (char *) prec, pbuflen, pbuf);
      put_bytes(&alpha, base++, 1, (char *) prec+1, pbuflen, pbuf);
      put_bytes(&alpha, base++, 8, (char *) prec+1+1, pbuflen, pbuf);
      put_bytes(&alpha, base++, 2, (char *) prec+1+1+8, pbuflen, pbuf);
      memcpy(&dummy_short, (char *)prec+1+1+8, 2);
      put_bytes(&alpha, base++, dummy_short, (char *) prec+1+1+8+2, pbuflen,
pbuf);
      break;

    case(AUT_RETURN64) :
      /* token ID | process error | process value
       *  1 byte      1 byte          8 byte
       */
      base = token_table[k].base_id;
      put_bytes(&alpha, base++, 1, (char *) prec, pbuflen, pbuf);
      put_bytes(&alpha, base++, 1, (char *) prec+1, pbuflen, pbuf);
      put_bytes(&alpha, base++, 8, (char *) prec+1+1, pbuflen, pbuf);
      break;

    case(AUT_ATTR64) :
      /* token ID | file mode | owner UID | owner GID | file system ID
       *  1 byte     4 bytes     4 bytes     4 bytes       4 bytes
          | file inode ID | device ID
              8 bytes       8 bytes
       */

      base = token_table[k].base_id;
      put_bytes(&alpha, base++, 1, (char *) prec, pbuflen, pbuf);
      put_bytes(&alpha, base++, 4, (char *) prec+1, pbuflen, pbuf);
      put_bytes(&alpha, base++, 4, (char *) prec+1+4, pbuflen, pbuf);
      put_bytes(&alpha, base++, 4, (char *) prec+1+4+4, pbuflen, pbuf);
      put_bytes(&alpha, base++, 4, (char *) prec+1+4+4+4, pbuflen, pbuf);
      put_bytes(&alpha, base++, 8, (char *) prec+1+4+4+4+4, pbuflen, pbuf);
      put_bytes(&alpha, base++, 8, (char *) prec+1+4+4+4+4+8, pbuflen, pbuf);
      break;

    case(AUT_HEADER64) :
      /* token ID | byte count | version # | event ID | ID modifier
       *  1 byte     4 bytes     1 byte     2 bytes      2 bytes
          | seconds of time | milisec of time
              8 bytes            8 bytes
       */
      base = token_table[k].base_id;
      put_bytes(&alpha, base++, 1, (char *) prec, pbuflen, pbuf);
      put_bytes(&alpha, base++, 4, (char *) prec+1, pbuflen, pbuf);
```

```
        put_bytes(&alpha, base++, 1, (char *) prec+1+4, pbuflen, pbuf);
        put_bytes(&alpha, base++, 2, (char *) prec+1+4+1, pbuflen, pbuf);
        put_bytes(&alpha, base++, 2, (char *) prec+1+4+1+2, pbuflen, pbuf);
        put_bytes(&alpha, base++, 8, (char *) prec+1+4+1+2+2, pbuflen, pbuf);
        put_bytes(&alpha, base++, 8, (char *) prec+1+4+1+2+2+8, pbuflen, pbuf);
        break;

    case(AUT_SUBJECT64) :
        /* token ID | audit ID | user ID | group ID | ru ID | rg ID | proc ID
         *  1 byte     4 bytes   4 bytes   4 bytes   4 bytes 4 bytes  4 bytes
            | sess ID | device ID | machine ID
              4 bytes    8 bytes     4 bytes
         */
        base = token_table[k].base_id;
        put_bytes(&alpha, base++, 1, (char *) prec, pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1, pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1+4 , pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1+4+4 , pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1+4+4+4 , pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1+4+4+4+4 , pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1+4+4+4+4+4 , pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1+4+4+4+4+4+4 , pbuflen, pbuf);
        put_bytes(&alpha, base++, 8, (char *) prec+1+4+4+4+4+4+4+4 , pbuflen,
pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1+4+4+4+4+4+4+4+8 , pbuflen,
pbuf);
        break;

    case(AUT_SERVER64) :
        exit(1);
        break;

    case(AUT_PROCESS64) :
        /* token ID | audit ID | user ID | group ID | ru ID | rg ID | proc ID
         *  1 byte     4 bytes   4 bytes   4 bytes   4 bytes 4 bytes  4 bytes
            | sess ID | device ID | machine ID
              4 bytes    8 bytes     4 bytes
         */
        base = token_table[k].base_id;
        put_bytes(&alpha, base++, 1, (char *) prec, pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1, pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1+4 , pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1+4+4 , pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1+4+4+4 , pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1+4+4+4+4 , pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1+4+4+4+4+4 , pbuflen, pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1+4+4+4+4+4+4 , pbuflen, pbuf);
        put_bytes(&alpha, base++, 8, (char *) prec+1+4+4+4+4+4+4+4 , pbuflen,
pbuf);
        put_bytes(&alpha, base++, 4, (char *) prec+1+4+4+4+4+4+4+4+8 , pbuflen,
pbuf);
```

```
        break;

    case(AUT_OTHER_FILE64) :
        exit(1);
        break;

    default :
        exit(1);
    }
  }

  *((int *) *pbuf) = alpha - *pbuf;
}
```

Annexe D

Code source de l'adaptateur de format TCPlogger

```
#include <sys/time.h> // For struct timeval
#include <stdio.h> // For *FILE and associated functions
#include <signal.h> // For signals management
#include <netinet/in.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include "msgs.h"
#include "netio.h"

#define NADF_BUFFER_SIZE 1024 // Size of the buffer where the Tcplogger record
will be converted in NADF
#define IDLGSZ (2*sizeof(u_short)) // Size of the ID field in an NADF record
#define CHUNK 1024 // Size for buffer extension

struct tcplogger_record {
  struct timeval tm; // Time of request
  u_long ipsrcaddr; // IP source address
  u_long ipdstaddr; // IP destination address
  u_long tcpseqno; // TCP sequence number
  u_short tcpsrcport; // TCP source port
  u_short tcpdstport; // TCP destination port
};

FILE* tcp_pipe; // The file handler used for the pipe to Tcplogger

char crtTimeStp[16], end[16];

struct sockaddr_in mstr_addr; // Socket address to talk to master
int mstr_addrlen; // It's length
int mstr_sock; // socket to talk to master evaluator

int mysock; // socket to talk to slaves
struct sockaddr_in myaddr; // socket address to talk to slaves
```

```
int dsock; // socket to talk to ASAX deamon

int tcploggerisrunning = 0;

extern tsslot *slv_list; /* from netio.c */

/* Pre :
 * - 'native_record' is a tcplogger_struct structure containing a native Tcplogger record.
 * - 'NADF_record' points to an allocated buffer able to receive an NADF record. It's size
must be big enough to handle all the fields required to describe a TCPLogger record plus
some informations which will be added later (typically, the name of the host).
 *
 * Post :
 * - 'native_record' is unchanged.
 * - The area pointed by 'NADF_record' contains the result of the conversion of
'native_record' into an NADF record.
 * - It returns the size of the NADF record or -1 in case of error.
 */

int NADF_convert(struct tcplogger_record native_record, char *NADF_record) {

  u_long NADF_size;
  u_short field_size;
  int id;
  char* ptr;
  u_long dummy_long;
  u_short dummy_short;
  int buflen = NADF_BUFFER_SIZE;

  ptr = NADF_record;

  /* An NADF record starts with an 4-bytes integer which is the size of the record (*with*
this heading integer). Since a converted TCPLogger record will always have the same size,
we can compute it in advance : it will be the size of each field present in the original
TCPLogger record plus 4 bytes for each field. These 4 bytes are the header present in each
NADF field. These 4 bytes code the ID number and the size of the field.
   */

  NADF_size = 60; // It's : 4+sizeof(seconds) + 4+sizeof(msecs) + 4+sizeof(u_long) +
4+sizeof(u_long) + 4+sizeof(u_long)
                  // + 4+sizeof(u_long) + 4+sizeof(u_long); PLUS THE
HEADING INTEGER
  memcpy(ptr, &NADF_size, 4);

  ptr += 4;


  id = 9000;

  put_bytes(&ptr, id++, 4, (char *)&native_record.tm.tv_sec, &buflen,
&NADF_record); // seconds of time
```

```
  put_bytes(&ptr, id++, 4, (char *)&native_record.tm.tv_usec, &buflen,
&NADF_record); // miliseconds of time
  put_bytes(&ptr, id++, 4, (char *)&native_record.ipsrcaddr, &buflen,
&NADF_record); // IP source address
  put_bytes(&ptr, id++, 4, (char *)&native_record.ipdstaddr, &buflen,
&NADF_record); // IP destination address
  put_bytes(&ptr, id++, 4, (char *)&native_record.tcpseqno, &buflen,
&NADF_record); // Sequence number
  dummy_long = native_record.tcpsrcport;
  put_bytes(&ptr, id++, 4, (char *)&dummy_long, &buflen, &NADF_record); // TCP
source port
  dummy_long = native_record.tcpdstport;
  put_bytes(&ptr, id++, 4, (char *)&dummy_long, &buflen, &NADF_record); // TCP
dest port


  return NADF_size;



/* Pre :
 * - 'f' is a valid file handler which is a pipe to a running copy of Tcplogger.
 * - 'native_record' points to a struct tcplogger_record.
 *
 * Post :
 * - a Tcplogger record is read from 'f' and is put into '*native_record'.
 * - returns the size of the native record read or -1 in case of error.
 */
int readnative(FILE* f, struct tcplogger_record * native_record) {

  int rc;

  rc = fread(native_record, sizeof(struct tcplogger_record), 1, f);

  if (ferror(f) != 0) return(-1); else return(sizeof(struct tcplogger_record));
}

/* Pre :
 * - None.
 *
 * Post : - Returns the result of a popen() of Tcplogger
 */
FILE* launchtcplogger() {
  FILE* f;

  f = popen("/etc/security/audit/ids/NETLOG1.2/bin/tcplogger -b","r");

  return f;
}

void catch(int sig) {
```

```
  fprintf(stderr, "[Netlog Supplier] Signal : requested termination.\n");
  fprintf(stderr, " Cause : caught signal %d.\n", sig);
  bailout();
}

void evilsig(int sig) {
  fprintf(stderr, "[Netlog Supplier] Signal : fatal signal %d caugh.\n", sig);
  (void)signal(SIGILL, SIG_DFL);
  (void)signal(SIGFPE, SIG_DFL);
  (void)signal(SIGBUS, SIG_DFL);
  (void)signal(SIGSEGV, SIG_DFL);
  (void)signal(SIGSYS, SIG_DFL);
  bailout();
}

int main() {
  int rc; // Used to store Return Code of some functions.

  struct tcplogger_record native_record; // Will contain a record from Tcplogger
  char NADF_record[NADF_BUFFER_SIZE]; // Will contain the record from
Tcplogger converted into NADF



  char *sptr, *eptr; /* to traverse crtTimeStp and end */
  int msg;
  int fd;
  int diff;

  (void)signal(SIGINT, catch);
  (void)signal(SIGHUP, catch);
  (void)signal(SIGQUIT, catch);
  (void)signal(SIGTERM, catch);
  (void)signal(SIGPIPE, catch);
  (void)signal(SIGILL, evilsig);
  (void)signal(SIGFPE, evilsig);
  (void)signal(SIGBUS, evilsig);
  (void)signal(SIGSEGV, evilsig);
  (void)signal(SIGSYS, evilsig);

  /* make socket and listen to slaves */

  mysock = makesock(&myaddr);

  if (mysock < 0) {
    fprintf(stderr, "[Netlog Supplier] Error : cannot make socket.\n");
    fprintf(stderr, " Check : makesock.\n");
    bailout();
  }

  /* send socket address and its length to asaxd */
```

```
  snd(0, PEER_MSG, "%d%k", sizeof(myaddr), myaddr); // 0 is stdout
  while ( (dsock = acceptslave(mysock)) <= 0);


  while (1) {
    tsslot *sl;
    msg = probemulti(&sl);


    if (msg == NO_MSG)
      continue;
  swit :

    switch (msg) {

    case ARG_MSG :
      sptr = &crtTimeStp[0];
      eptr = &end[0];
      getarg_msg(sl→slvsock, &mstr_addr, &mstr_addrlen, sptr, eptr);

      mstr_sock = connectToMaster(&mstr_addr, mstr_addrlen, 0);
      fcntl(mstr_sock, F_SETFL, O_NONBLOCK);


      break;

    case REPOS_MSG :
      getrepos_msg(sl→slvsock, crtTimeStp, end);
      break;

    case SUPPLY_MSG :
      if ( !tcploggerisrunning) {
        tcp_pipe = launchtcplogger();

        if (tcp_pipe == NULL) {
          fprintf(stderr, "[Netlog Supplier] Error : unable to launch
tcplogger.\n");
          fprintf(stderr, " Check : launchtcplogger.\n");
          bailout();
        }


        rc = readnative(tcp_pipe, &native_record);

        if (rc < 0) {
          fprintf(stderr, "[Netlog Supplier] Error : problem while reading
tcplogger data.\n");
          fprintf(stderr, " Check : readnative.\n");
          bailout();
        }
```

```
        rc = NADF_convert(native_record, NADF_record);

        if (rc < 0) {
          fprintf(stderr, "[Netlog Supplier] Error : unable to convert Netlog
data into NADF.\n");
          fprintf(stderr, " Cause : not enough memory to reallocate NADF
buffer.\n");
          fprintf(stderr, " Check : NADF_convert and put_bytes.\n");
          bailout();
        }
      }

      diff = -1;
      while (diff < 0 && rc >= 0) {
        snd(mstr_sock, RECORD_MSG, "%R", NADF_record);
        msg = NO_MSG;
        rc=fread_NADF_Probe(tcp_pipe, &NADF_record, &msg);
        if (msg == NO_MSG && rc >= 0) {
          //getTimeStr(crtTimeStp, getTimeStp(NADF_record));
          //diff = strcmp(crtTimeStp, end);
        } else if (msg!= NO_MSG) {
          goto swit;
        } else { /* (rc < 0) */
          fprintf(stderr, "error reading :%d", rc);
          bailout();
        }

      }
      if (diff >= 0) {
        bailout();
      }

      break;

    case END_MSG :
      bailout();

    default :
      fprintf(stderr, "[Netlog Supplier] Error : unexpexted control
message..\n");
      fprintf(stderr, " Cause : received control message %d.\n",msg);
      fprintf(stderr, " Check : main.\n");
      bailout();


    } /* switch (msg) */
  } // while(1)
  pclose(tcp_pipe);
  exit(0);
}
```

```
fread_NADF_Probe(FILE* mypipe, char *NADF, int *from)
{
      int rc, msg;
      struct tcplogger_record native_record;
      tsslot *sl;
      rc = 0;
      msg = NO_MSG;
      // while (1) {
         msg = probemulti(&sl);
         if (msg && msg!= NO_MSG) {
                  *from = msg;
                  return rc;
         }

         rc = readnative(mypipe, &native_record);

         if (rc < 0) {
           fprintf(stderr, "[Netlog Supplier] Error : problem while reading
tcplogger data.\n");
           fprintf(stderr, " Check : readnative.\n");
           bailout();
         }

         rc = NADF_convert(native_record, NADF);

         if (rc < 0) {
           fprintf(stderr, "[Netlog Supplier] Error : unable to convert Netlog
data into NADF.\n");
           fprintf(stderr, " Cause : not enough memory to reallocate NADF
buffer.\n");
           fprintf(stderr, " Check : NADF_convert and put_bytes.\n");
           bailout();
         }


         //}
      *from = NO_MSG;

       return(rc);
}


int bailout() {
  int msg;

  fprintf(stderr, "[Netlog Supplier] : terminating...\n");
  if (tcp_pipe!= NULL) {
    fprintf(stderr, " closing pipe to Tcplogger...\n");
    pclose(tcp_pipe);
```

```
  }
  snd(mstr_sock, END_MSG, "");
  snd(dsock, END_MSG, "");
  if ( (msg = msgtype(dsock)) < 0) {
    perror("msgtype");
    exit(1);
  }
  if (msg == END_MSG) {
    exit(1);
  }
  while (slv_list)
    free_sl_slot(slv_list);
  shutdown(mstr_sock, 2);
  fprintf(stderr, "[Netlog Supplier] : End of execution...\n");
  exit(0);
}
```